ЛАБОРАТОРНАЯ РАБОТА №4

Тема: Разработка драйвера для ОС Linux (Ubuntu)

Цель работы: Изучить основы разработки драйверов для ядра Linux с использованием языка программирования C, включая настройку окружения, создание драйвера и его тестирование.

Цель работы обуславливает постановку и решение следующих **задач**:

1. Настроить окружение для разработки драйвера на языке C для ОС Linux (Ubuntu).
2. Разработать и протестировать демо-драйвер my_sum на языке C.
2.1. Освоить основные принципы программирования драйвера.
2.2. Реализовать работу с мьютексами в драйвере.
3. Выполнить индивидуальное задание, закрепляющее на практике полученные знания (номер задания соответствует номеру студента по журналу; если этот номер больше, чем максимальное число заданий, тогда вариант задания вычисляется по формуле: номер по журналу % максимальный номер задания, где % — остаток от деления).
3.1. Добавить логирование операций с использованием встроенных средств ядра Linux.
3.2. Создать структуру для управления ресурсами драйвера, включая выделение и освобождение памяти.
3.3. Протестировать работу разработанного драйвера на производительность при высоком числе одновременных запросов.
4. Составить отчет, включающий код разработанного драйвера, результаты тестирования и выводы.

Условия подготовки и проведения экспериментов для виртуальной машины

Внимание! Работа в режиме суперпользователя с системой Линукс сопряжена с риском потери данных и дестабилизации ОС. Перед критическими экспериментами настоятельно рекомендуется сохранять снепшоты (моментальные снимки состояния системы в определенные моменты времени) для возможности резервного восстановления. Снепшоты позволяют сохранить текущее состояние ОС, файлов, конфигураций и всех данных виртуальной машины.

С помощью утилиты vmrun, которая находится обычно после инсталляции VMware Player в папке C:\Program Files (x86)\VMware\VMware Player\ для активной виртуальной машины делаем снепшот с некоторым уникальным именем из командной строки:

```
vmrun -T ws snapshot "C:\Users\Aleks\Documents\Virtual Machines\Ubuntu 64-
bit\Ubuntu 64-bit.vmx" "Snapshot 1"
```

В случае аварии для выключенной виртуальной машины восстанавливаем исходное состояние:

```
vmrun -T ws revertToSnapshot "C:\Users\Aleks\Documents\Virtual Machines\Ubuntu 64-
bit\Ubuntu 64-bit.vmx" "Snapshot 1"
```

Ход выполнения лабораторной работы

1. Настройка окружения

Установите необходимые пакеты:

```
sudo apt update && sudo apt install gcc build-essential linux-headers-$(uname -r)
```

2. Создание драйвера для подсчета суммы чисел.

```
mkdir ~/my_sum_driver
cd ~/my_sum_driver
```

Файл ~/my_sum/my_sum.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/mutex.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("A.M.Ostrowski");
MODULE_DESCRIPTION("A driver for summing numbers via /dev/my_sum");
MODULE_VERSION("1.0");

#define DEVICE_NAME "my_sum"
#define BUFFER_SIZE 256

static int major_number;
static int current_sum = 0; // Сумма
static char buffer[BUFFER_SIZE];
static size_t buffer_size = 0;
static struct mutex sum_mutex; // Мьютекс

static int device_open(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "my_sum: Device opened\n");
    return 0;
}

static int device_release(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "my_sum: Device closed\n");
    return 0;
}

static ssize_t device_read(struct file *filep, char *user_buffer, size_t len,
loff_t *offset) {
    int ret;

    if (*offset > 0)
        return 0;

    mutex_lock(&sum_mutex);
```

```
    buffer_size = snprintf(buffer, BUFFER_SIZE, "%d\n", current_sum);
    mutex_unlock(&sum_mutex);

    ret = copy_to_user(user_buffer, buffer, buffer_size);
    if (ret == 0) {
        printk(KERN_INFO "my_sum: Sent sum %d to user\n", current_sum);
        *offset += buffer_size;
        return buffer_size;
    } else {
        printk(KERN_ERR "my_sum: Failed to send data to user\n");
        return -EFAULT;
    }
}

static ssize_t device_write(struct file *filep, const char *user_buffer, size_t
len, loff_t *offset) {
    char input[BUFFER_SIZE] = {0};
    int number;

    if (len >= BUFFER_SIZE) {
        printk(KERN_ERR "my_sum: Input is too long\n");
        return -EINVAL;
    }

    if (copy_from_user(input, user_buffer, len)) {
        printk(KERN_ERR "my_sum: Failed to copy data from user\n");
        return -EFAULT;
    }

    input[len] = '\0';
    if (kstrtoint(input, 10, &number) != 0) {
        printk(KERN_ERR "my_sum: Invalid number format\n");
        return -EINVAL;
    }

    mutex_lock(&sum_mutex);
    if (!number)
        current_sum = 0;
    else
        current_sum += number;
    printk(KERN_INFO "my_sum: Added %d, new value is %d\n", number, current_sum);
    mutex_unlock(&sum_mutex);

    return len;
}

static struct file_operations fops = {
    .open = device_open,
    .read = device_read,
    .write = device_write,
    .release = device_release,
};

static int __init my_sum_init(void) {
    major_number = register_chrdev(0, DEVICE_NAME, &fops);
    if (major_number < 0) {
        printk(KERN_ERR "my_sum: Failed to register device\n");
```

```
        return major_number;
    }

    mutex_init(&sum_mutex);
    printk(KERN_INFO "my_sum: Registered with major number %d\n", major_number);
    return 0;
}

static void __exit my_sum_exit(void) {
    mutex_destroy(&sum_mutex);
    unregister_chrdev(major_number, DEVICE_NAME);
    printk(KERN_INFO "my_sum: Device unregistered\n");
}

module_init(my_sum_init);
module_exit(my_sum_exit);
```

Этот код представляет собой драйвер ядра Linux для устройства /dev/my_sum, позволяющего суммировать числа, записанные пользователем, и считывать их результат.

Устройство /dev/my_sum регистрируется с помощью вызова register_chrdev. Мьютекс (sum_mutex) инициализируется для синхронизации доступа к общей переменной current_sum. Функция device_write позволяет пользователю записывать числа в устройство. Если записанное значение равно 0, сумма сбрасывается; в противном случае новое значение добавляется к текущей сумме. Функция device_read возвращает текущее значение суммы через интерфейс устройства. Мьютекс защищает доступ к переменной current_sum, чтобы предотвратить состояние гонки, когда несколько процессов одновременно обращаются к устройству. Функции mutex_lock и mutex_unlock используются для блокировки и разблокировки доступа к общей переменной. Устройство открывается через device_open, а закрывается через device_release. Для записи используется device_write, где данные пользователя копируются в ядро с помощью copy_from_user. После обработки данные добавляются к текущей сумме. Для чтения используется device_read, где сумма копируется в пространство пользователя через copy_to_user. Ввод ограничен буфером размером BUFFER_SIZE (256 байт). Если пользователь пытается записать данные, превышающие этот размер, операция завершится ошибкой. Если переданное значение не является корректным числом, ввод отклоняется. Ошибки чтения и записи логируются через printk с уровнями KERN_INFO и KERN_ERR.

3. Настройка Makefile:

```
obj-m += my_sum.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
```

4. Собираем драйвер (из корневой директории исходного кода ядра Linux):

```
make clean
make -j$(nproc)
```

5. Загружаем модуль:

```
sudo insmod my_sum.ko
```

Проверяем, загружен ли драйвер:

```
lsmod | grep my_sum
```

Проверяем лог:

```
sudo dmesg | grep "my_sum"
```

Создаём файл-устройство:

```
sudo mknod /dev/my_sum c <major> 0
```

Где <major> — номер, указанный в выводе dmesg.

```
echo 2 | sudo tee /dev/my_sum
echo 3 | sudo tee /dev/my_sum
```

Проверяем результат:

```
cat /dev/my_sum
```

```
kon@kon:~/my_sum_driver$ make -j$(nproc)
make -C /lib/modules/6.8.0-49-generic/build M=/home/kon/my_sum_driver modules
make[1]: warning: jobserver unavailable: using -j1.  Add '+' to parent make rule.
make[1]: Entering directory '/usr/src/linux-headers-6.8.0-49-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0
  You are using:           gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0
  CC [M]  /home/kon/my_sum_driver/my_sum.o
  MODPOST /home/kon/my_sum_driver/Module.symvers
  CC [M]  /home/kon/my_sum_driver/my_sum.mod.o
  LD [M]  /home/kon/my_sum_driver/my_sum.ko
  BTF [M] /home/kon/my_sum_driver/my_sum.ko
Skipping BTF generation for /home/kon/my_sum_driver/my_sum.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.8.0-49-generic'
kon@kon:~/my_sum_driver$ sudo insmod my_sum.ko
kon@kon:~/my_sum_driver$ lsmod | grep my_sum
my_sum                 12288  0
kon@kon:~/my_sum_driver$ sudo dmesg | grep "my_sum"
[  567.299723] my_sum: loading out-of-tree module taints kernel.
[  567.299822] my_sum: module verification failed: signature and/or required key missing - tainting kernel
[  567.301552] my_sum: Registered with major number 240
kon@kon:~/my_sum_driver$ sudo mknod /dev/my_sum c 240 0
kon@kon:~/my_sum_driver$ echo 2 | sudo tee /dev/my_sum
2
kon@kon:~/my_sum_driver$ echo 3 | sudo tee /dev/my_sum
3
kon@kon:~/my_sum_driver$ cat /dev/my_sum
5
kon@kon:~/my_sum_driver$
```

6. Нагрузочное тестирование драйвера (stress_test_my_sum.sh):

```bash
#!/bin/bash

DEVICE="/dev/my_sum"
if [ ! -e "$DEVICE" ]; then
    echo "Ошибка: Устройство $DEVICE не существует."
    exit 1
fi

run_test() {

    local input="$1"
    local expected_sum="$2"

    # Запись данных в устройство
    echo "$input" | sudo tee "$DEVICE" > /dev/null
    if [ $? -ne 0 ]; then
        echo "Ошибка записи: $input"
        exit 2
    fi

    # Чтение результата из устройства
    local result=$(cat "$DEVICE")
    if [ $? -ne 0 ]; then
        echo "Ошибка чтения результата"
        exit 3
    fi

    # Проверка результата
    if [ "$result" -eq "$expected_sum" ]; then
```

```
        echo "Тест пройден: Ввод = '$input', Ожидаемая сумма = $expected_sum,
Полученная = $result"
    else
        echo "Ошибка: Ввод = '$input', Ожидаемая сумма = $expected_sum, Полученная
= $result"
        exit 4
    fi
}

echo "Начинаем нагрузочное тестирование драйвера $DEVICE..."

# Сбрасываем сумму перед началом тестирования
echo 0 | sudo tee "$DEVICE" > /dev/null

# Последовательное тестирование
for i in {1..50}; do
    run_test 3 $((i * 3))
done

# Сбрасываем сумму перед параллельным тестированием
echo 0 | sudo tee "$DEVICE" > /dev/null

echo "Параллельное тестирование..."
for i in {1..50}; do
    (echo "$i" | sudo tee /dev/my_sum > /dev/null) &
done
wait

PR=$(cat "$DEVICE")
if [ $? -ne 0 ]; then
    echo "Ошибка чтения результата"
    exit 5
fi

if [ "$PR" -eq 1275 ]; then
    echo "Параллельный тест пройден"
else
    echo "Ошибка при параллельных запросах: Ожидаемое значение = 1275, Полученное
значение = $PR"
    exit 6
fi

echo "Тестирование завершено успешно!"
exit 0
```

7. Удаляем драйвер (как завершение цикла активности ПО):

```
sudo rmmod my_sum
```

Индивидуальные задания

1. Разработать драйвер (по типу /dev/urandom) для генерирования псевдослучайных вещественных чисел с распределением Пауссона. Подтвердить характер распределения построением соответствующей кривой.

2. Разработать драйвер, логирующий символы со стандартного потока ввода (клавиатуры). Использовать register_keyboard_notifier, чтобы получать уведомления о событиях клавиатуры. При выгрузке драйвера вызвать unregister_keyboard_notifier, чтобы отключить обработчик.

3. Реализовать драйвер для виртуального сетевого устройства, которое принимает входящие пакеты и отправляет обратно тому же отправителю ("эхо").

4. Разработать драйвер устройства для считывания температуры с виртуального датчика. Реализовать возможность чтения текущей температуры из /dev/virtual_temp, где температура обновляется каждую секунду. Проверить корректность работы с помощью утилиты cat или tail -f.

5. Реализовать драйвер для подсчёта количества открытых дескрипторов файла в системе. Драйвер должен возвращать число открытых дескрипторов при чтении из файла устройства.

6. Разработать драйвер, реализующий простую базу данных в оперативной памяти. Реализовать базовые операции: запись и чтение строковых значений по целочисленному ключу.

7. Создать драйвер виртуального устройства /dev/stack, реализующего глобальный стек для строковых данных. Реализовать команды push, pop и peek.

8. Создать драйвер виртуального устройства /dev/queue, реализующего глобальную очередь для строковых данных. Реализовать команды enqueue, dequeue и peek.

9. Разработать драйвер, логирующий события манипулятора типа мышь. Реализовать регистрацию событий перемещения, нажатия кнопок и прокрутки колеса.

10. Разработать драйвер-эмулятор клавиатуры. Драйвер должен отправлять случайно сгенерированные символы в системный буфер с заданной периодичностью.

11. Разработать драйвер, который работает через framebuffer (fbdev) и позволяет рисовать на совместимом устройстве простейшие геометрические фигуры, такие как круг, треугольник и квадрат. Координаты фигур и их цвет драйвер выбирает случайным образом.

```
echo "square" > /dev/fb0
echo "circle" > /dev/fb0
echo "triangle" > /dev/fb0
```

12. Разработать драйвер-эмулятор видео-камеры (например, с использованием v4l2) для вывода статического изображения. Протестировать работоспособность со стандартным видеоплеером.

13. Реализовать драйвер /dev/watchdog_timer, который выполняет следующие функции. Драйвер должен инициализировать watchdog таймер, который перезапускается при записи в устройство. Если запись в устройство не выполняется в течение заданного времени (например, 30 секунд), драйвер генерирует сигнал сбоя.

14. Разработать драйвер-эмулятор для устройства "виртуальная батарея". Драйвер должен предоставлять интерфейс /dev/virtual_battery, в котором отображается уровень заряда батареи (от 0% до 100%) и состояние зарядки (заряжается или разряжается). Уровень заряда изменяется каждые 5 секунд в зависимости от текущего состояния. Проверить работу с помощью утилиты cat и реализовать возможность переключения состояний зарядки через echo "charge" > /dev/virtual_battery и echo "discharge" > /dev/virtual_battery.

15. Создать драйвер устройства /dev/morse_code, который преобразует переданную строку в код Морзе. Результат преобразования возвращается при чтении файла устройства.

16. Реализовать драйвер /dev/virtual_keyboard_mapper, который позволяет переназначать клавиши клавиатуры. Например, при записи в устройство строки "a -> b", нажатие клавиши "a" будет интерпретироваться как нажатие "b". Проверить работу через стандартный ввод.

17. Создать драйвер устройства /dev/image_converter, который принимает изображения в формате PNG и конвертирует их в формат JPEG. Результат сохраняется в директории /tmp.

18. Создать драйвер /dev/process_brake для торможения процессов, который замедляет выполнение процессов с указанными PID. Замедление задается в процентах при записи команды.

Контрольные вопросы

1. Какие преимущества язык C обеспечивает в разработке драйверов ядра Linux?
2. Какие средства синхронизации предоставляет C для работы с многопоточными процессами в ядре?
3. Как с помощью мьютексов предотвращается возникновение гонок данных в драйверах?
4. Как используется асинхронное программирование в контексте драйверов?
5. Какие шаги необходимо выполнить для настройки окружения разработки драйвера на языке C для ядра Linux?
6. Какие преимущества использования встроенных средств логирования ядра Linux при разработке драйверов?
7. Как проверить корректность работы драйвера после его загрузки в систему?
8. Какие действия необходимо выполнить для создания файла-устройства, связанного с драйвером?
9. Как выполняется нагрузочное тестирование драйвера и какие сценарии необходимо учитывать?
10. Какие действия необходимо выполнить для безопасного удаления драйвера из ядра?
11. Какие типы виртуальных устройств могут быть реализованы на основе драйверов?
12. Как обеспечить стабильность и отказоустойчивость драйвера при интенсивной нагрузке?

13. Какие аспекты необходимо учитывать при разработке драйверов с использованием встроенных методов ядра для управления ресурсами?
14. Какие ограничения и особенности необходимо учитывать при разработке драйверов?
15. Как реализовать обработку событий с использованием сигналов в драйверах ядра Linux?
16. Объясните, как драйвер может взаимодействовать с процессами пользовательского пространства через механизмы сигнализации.