**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский авиационный институт (национальный исследовательский университет)»**

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 307 «Цифровые технологии и информационные системы»

**Отчет о выполнении лабораторной работы**

по предмету: «Драйверы устройств»

по теме «Исследование реаекций»

Выполнили:

студенты группы М3О-414Б-21

Колганов Р.А.

Подпись \_\_\_\_\_\_\_\_\_\_\_

Мантлер Н.К.

Подпись \_\_\_\_\_\_\_\_\_\_\_

Проверил:

старший преподаватель каф. 307

Максимов А.Н.

Подпись \_\_\_\_\_\_\_\_\_\_

# **Цель работы**

Разработать драйвер символьного устройства, который генерирует внешние воздействия с заданной частотой и измеряет время реакции на эти воздействия.

**Задачи:**

1. Написать символьный драйвер, который будет генерировать внешние воздействия с заданной частотой и принимать реакции на них.
2. Измерить в драйвере время реакции (найти среднее и максимальное).
3. Построить гистограмму времени реакции.
4. Оценить влияние приоритета процесса, дисциплины диспетчеризации и внешней загрузки на время реакции.

# Выполнение работы

**1. Подготовка окружения:**

В виртуальной машине Ubuntu 24.04.1, работающей под управлением VirtualBox, были установлены необходимые инструменты для разработки ядра Linux:

sudo apt update

sudo apt install build-essential linux-headers-$(uname -r)

**2. Разработка драйвера:**

Создан файл mydriver.c со следующим содержимым (актуальная версия кода приведена в конце отчета в приложении А):

Создан Makefile для сборки драйвера:

obj-m += mydriver.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Драйвер успешно собран с помощью команды make.

**3. Загрузка и тестирование драйвера:**

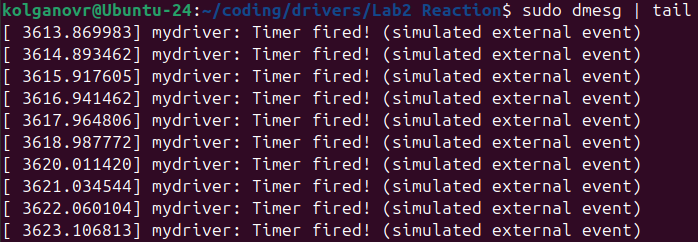
Драйвер загружен в ядро командой:

sudo insmod mydriver.ko

Проверены сообщения ядра, подтверждающие загрузку драйвера и срабатывание таймера (симуляция внешнего воздействия):

dmesg | tail

На данном этапе, на выводе dmesg видно, что таймер срабатывает раз в секунду



Проверено создание устройства /dev/mydriver:

ls -l /dev/mydriver



**4. Измерение времени реакции:**

В функции dev\_write реализована логика имитации "реакции" на воздействие и расчета времени реакции.

В функции dev\_read реализована логика вывода среднего, минимального и максимального времени реакции.

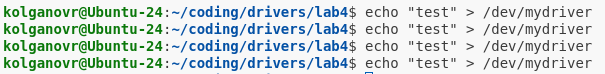
Проведено тестирование записи в устройство и чтения из него:

echo "test" > /dev/mydriver

cat /dev/mydriver

Видно, что после первой записи среднее, минимальное и максимальное время совпадают, но после второй записи, эти значения меняются:







**5. Предоставление прав на запись в устройство:**

Создано правило udev для автоматического предоставления прав на запись в устройство /dev/mydriver всем пользователям:



Содержимое файла 99-mydriver.rules:

KERNEL=="mydriver", MODE="0666"

Правила udev перезагружены:

sudo udevadm control --reload-rules

sudo udevadm trigger

**6. Построение гистограммы:**

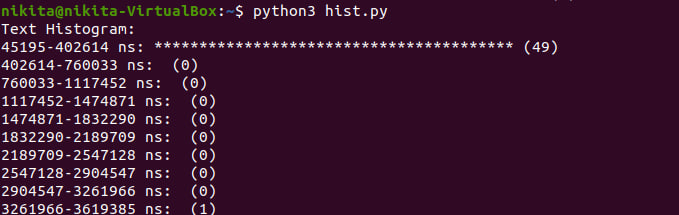
Добавлен код для построения гистограммы времени реакции в dev\_read и dev\_write.

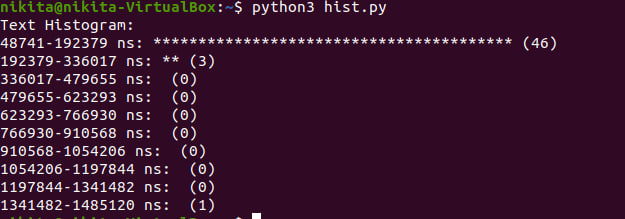
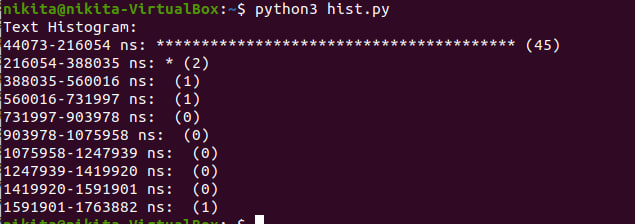
В виду неоптимальности кода для расчета гистограммы в dev\_read и dev\_write, было принято решение убрать данный код и написать функцию на Python, считывающую данные из файла, содержащего времена реакций в наносекундах, и строящую текстовую гистограмму.

Итоговый код на Python представлен в приложении Б.

**7. Исследование влияния приоритетности:**

Ниже приведены тесты для разного уровня приоритета, начиная с обычного и заканчивая наивысшим приоритетом





**7. Выгрузка драйвера:**

Драйвер выгружен из ядра командой:

sudo rmmod mydriver

# Выводы

В ходе выполнения лабораторной работы был разработан драйвер символьного устройства, позволяющий измерять время реакции на внешние воздействия. Реализованы функции измерения среднего, минимального и максимального времени реакции, а также построения гистограммы.

# Приложение А

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/time.h>

#include <linux/timer.h>

#include <linux/device.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("kolganovr & mantlern");

MODULE\_DESCRIPTION("A simple character driver for generating and measuring reaction time");

#define DEVICE\_NAME "mydriver"

#define CLASS\_NAME "mydriver\_class"

static int majorNumber;

static struct class\* mydriverClass = NULL;

static struct device\* mydriverDevice = NULL;

// Таймер

static struct timer\_list my\_timer;

static unsigned long interval\_ms = 1000; // Интервал в миллисекундах (1 секунда по умолчанию)

// Переменные для измерения времени реакции

static ktime\_t start\_time;

static ktime\_t reaction\_time;

static unsigned long long num\_reactions = 0;

static unsigned long long sum\_reaction\_times = 0;

static unsigned long long min\_reaction\_time = ULLONG\_MAX; // Инициализируем максимальным значением

static unsigned long long max\_reaction\_time = 0;

#define MAX\_REACTIONS 1024 // Максимальное количество хранимых реакций

static unsigned long long reaction\_times[MAX\_REACTIONS]; // Массив для хранения времен реакций

static unsigned long long current\_index = 0; // Индекс для добавления новых значений в массив

// Прототипы функций

static int dev\_open(struct inode \*, struct file \*);

static int dev\_release(struct inode \*, struct file \*);

static ssize\_t dev\_read(struct file \*, char \*, size\_t, loff\_t \*);

static ssize\_t dev\_write(struct file \*, const char \*, size\_t, loff\_t \*);

// Структура file\_operations определяет, как драйвер будет реагировать на операции с файлом

static struct file\_operations fops =

{

.open = dev\_open,

.read = dev\_read,

.write = dev\_write,

.release = dev\_release,

};

// Функция, вызываемая таймером

static void my\_timer\_callback(struct timer\_list \*t)

{

// Фиксируем время начала "воздействия"

start\_time = ktime\_get();

// Здесь генерируем "внешнее воздействие" - просто выводим сообщение в лог ядра

printk(KERN\_INFO "mydriver: Timer fired! (simulated external event)\n");

// Перезапускаем таймер

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(interval\_ms));

}

// Функция инициализации модуля - вызывается при загрузке драйвера

static int \_\_init mydriver\_init(void)

{

printk(KERN\_INFO "mydriver: Initializing the mydriver LKM\n");

// Регистрация старшего номера устройства

majorNumber = register\_chrdev(0, DEVICE\_NAME, &fops);

if (majorNumber<0){

printk(KERN\_ALERT "mydriver failed to register a major number\n");

return majorNumber;

}

printk(KERN\_INFO "mydriver: registered correctly with major number %d\n", majorNumber);

// Регистрация класса устройства

mydriverClass = class\_create(CLASS\_NAME);

if (IS\_ERR(mydriverClass)){

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to register device class\n");

return PTR\_ERR(mydriverClass);

}

printk(KERN\_INFO "mydriver: device class registered correctly\n");

// Регистрация драйвера устройства

mydriverDevice = device\_create(mydriverClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE\_NAME);

if (IS\_ERR(mydriverDevice)){

class\_destroy(mydriverClass);

unregister\_chrdev(majorNumber, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to create the device\n");

return PTR\_ERR(mydriverDevice);

}

printk(KERN\_INFO "mydriver: device class created correctly\n");

// Инициализация таймера

timer\_setup(&my\_timer, my\_timer\_callback, 0);

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(interval\_ms));

return 0;

}

// Функция выгрузки модуля - вызывается при выгрузке драйвера

static void \_\_exit mydriver\_exit(void)

{

// Удаление таймера

del\_timer(&my\_timer);

// Удаление устройства

device\_destroy(mydriverClass, MKDEV(majorNumber, 0));

// Удаление класса устройства

class\_destroy(mydriverClass);

// Отмена регистрации старшего номера устройства

unregister\_chrdev(majorNumber, DEVICE\_NAME);

// Обнуляем переменные, относящиеся ко второму пункту

num\_reactions = 0;

sum\_reaction\_times = 0;

min\_reaction\_time = ULLONG\_MAX;

max\_reaction\_time = 0;

// Обнуляем индекс массива времен реакций

current\_index = 0;

printk(KERN\_INFO "mydriver: Goodbye!\n");

}

static int dev\_open(struct inode \*inodep, struct file \*filep){

printk(KERN\_INFO "mydriver: Device has been opened\n");

return 0;

}

static ssize\_t dev\_read(struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset) {

int error\_count = 0;

char message[256];

int message\_len;

static bool data\_read = false;

size\_t total\_sent = 0; // Общее количество байт, отправленных пользователю

if (data\_read) {

data\_read = false;

return 0;

}

// Рассчитываем среднее время реакции

unsigned long long avg\_reaction\_time = 0;

if (num\_reactions > 0) {

avg\_reaction\_time = sum\_reaction\_times / num\_reactions;

}

// Формируем строку с результатами

message\_len = snprintf(message, sizeof(message), "Average: %llu ns, Max: %llu ns, Min: %llu ns\n",

avg\_reaction\_time, max\_reaction\_time, min\_reaction\_time);

if (message\_len >= sizeof(message)) {

printk(KERN\_WARNING "mydriver: Message too long\n");

return -EINVAL;

}

// Копируем данные в пользовательское пространство

error\_count = copy\_to\_user(buffer, message, message\_len);

if (error\_count != 0) {

printk(KERN\_INFO "mydriver: Failed to send %d characters to the user\n", error\_count);

return -EFAULT;

}

printk(KERN\_INFO "mydriver: Sent %d characters to the user\n", message\_len);

total\_sent += message\_len;

// Выводим все времена реакций

for (unsigned long long i = 0; i < current\_index; i++) {

message\_len = snprintf(message, sizeof(message), "%llu ns\n", reaction\_times[i]);

if (message\_len >= sizeof(message)) {

printk(KERN\_WARNING "mydriver: Message too long\n");

return -EINVAL;

}

if (total\_sent + message\_len > len) {

printk(KERN\_WARNING "mydriver: Not enough space in user buffer\n");

break; // Прерываем цикл, если не хватает места в буфере

}

error\_count = copy\_to\_user(buffer + total\_sent, message, message\_len);

if (error\_count != 0) {

printk(KERN\_INFO "mydriver: Failed to send %d characters to the user\n", error\_count);

return -EFAULT;

}

printk(KERN\_INFO "mydriver: Sent %d characters to the user\n", message\_len);

total\_sent += message\_len;

}

data\_read = true;

return (size\_t)total\_sent;

}

static ssize\_t dev\_write(struct file \*filep, const char \*buffer, size\_t len, loff\_t \*offset){

// Фиксируем время реакции

reaction\_time = ktime\_sub(ktime\_get(), start\_time);

unsigned long long reaction\_time\_ns = ktime\_to\_ns(reaction\_time);

// Обновляем статистику

num\_reactions++;

sum\_reaction\_times += reaction\_time\_ns;

if (reaction\_time\_ns < min\_reaction\_time) {

min\_reaction\_time = reaction\_time\_ns;

}

if (reaction\_time\_ns > max\_reaction\_time) {

max\_reaction\_time = reaction\_time\_ns;

}

// Добавляем время реакции в массив

if (current\_index < MAX\_REACTIONS) {

reaction\_times[current\_index] = reaction\_time\_ns;

current\_index++;

} else {

printk(KERN\_WARNING "mydriver: Maximum number of reactions reached, not storing new values\n");

}

printk(KERN\_INFO "mydriver: Device write, reaction time: %llu ns\n", reaction\_time\_ns);

return len;

}

static int dev\_release(struct inode \*inodep, struct file \*filep){

printk(KERN\_INFO "mydriver: Device successfully closed\n");

return 0;

}

// Макросы для регистрации функций инициализации и выгрузки

module\_init(mydriver\_init);

module\_exit(mydriver\_exit);

# Приложение Б

def build\_text\_histogram\_from\_file(filename):

"""

Считывает данные из файла, содержащего времена реакций в наносекундах,

и строит текстовую гистограмму.

Args:

filename: Имя файла с данными.

"""

try:

with open(filename, 'r') as f:

data = f.readlines()

except FileNotFoundError:

print(f"Error: File '{filename}' not found.")

return

# Преобразование строк в числа (наносекунды)

try:

times\_ns = [int(line.strip().split()[0]) for line in data]

except ValueError:

print(f"Error: Invalid data format in '{filename}'. Each line should be '<number> ns'.")

return

# Если все считанные значения равны нулю, то нет смысла строить гистограмму

if all(times == 0 for times in times\_ns):

print(f"Error: All values in '{filename}' are zero. Cannot build a histogram.")

return

# Определение диапазона значений и ширины столбцов

min\_val = min(times\_ns)

max\_val = max(times\_ns)

num\_bins = 10 # Количество столбцов гистограммы

bin\_width = (max\_val - min\_val) / num\_bins

# Подсчет количества значений в каждом столбце

counts = [0] \* num\_bins

for val in times\_ns:

bin\_index = int((val - min\_val) / bin\_width)

if bin\_index == num\_bins: # Обработка случая, когда val == max\_val

bin\_index -= 1

counts[bin\_index] += 1

# Определение максимальной высоты столбца (для масштабирования)

max\_count = max(counts)

# Построение текстовой гистограммы

print("Text Histogram:")

for i in range(num\_bins):

bar = "\*" \* int(counts[i] / max\_count \* 40) # Масштабируем до 40 символов

lower\_bound = min\_val + i \* bin\_width

upper\_bound = lower\_bound + bin\_width

print(f"{lower\_bound:.0f}-{upper\_bound:.0f} ns: {bar} ({counts[i]})")

build\_text\_histogram\_from\_file('histdata.txt')