|  |  |
| --- | --- |
| ÐÐ¾Ð³Ð¾ÑÐ¸Ð¿ ÐÐÐ | 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 №2 по дисциплине

«Драйверы устройств»

по теме:

«Оценка задержки реакции системы на внешние воздействия»

Выполнили студенты  
группы: М3О-421Б-21  
Шульц Богдан   
Сунгатуллин Саид  
Олейник София  
Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил преподаватель:  
Максимов Алексей Николаевич

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Задания лабораторной работы

1. Написать символьный драйвер, который будет генерировать с заданной частотой внешние воздействия и принимать реакцию на них;
2. Померить в драйвере время реакции (найти среднее и максимальное);
3. Построить гистограмму времени реакции;
4. Оценить влияние:

а) Приоритета процесса;

б) Дисциплины диспетчеризации;

в) Внешней загрузки (HDD, сеть).

Выполнение задания

Написаны три программы:

1. Модуль ядра (символьный драйвер), который генерирует внешние воздействия каждые 100 мс. и считает время реакции (см. приложение А);
2. Приложение, которое взаимодействует с устройством каждые 10 мс. (см. приложение Б);

Были проведены следующие тесты на операционной системе Ubuntu 22.04:

1. В обычном режиме (см. рис. Рисунок 1):

Среднее время отклика = 2.00597 мс.

Максимальное время отклика = 41.16306 мс.

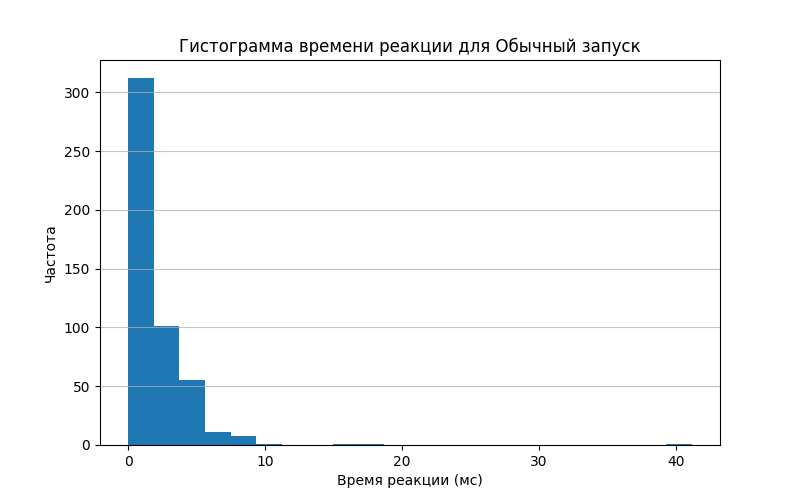


Рисунок 1 — Гистограмма времени реакции на событие в обычном режиме.

1. Высокий приоритет (значение приоритета ) (см. рис. Рисунок 2):

Среднее время отклика = 1.28338 мс.

Максимальное время отклика = 9.79093 мс.

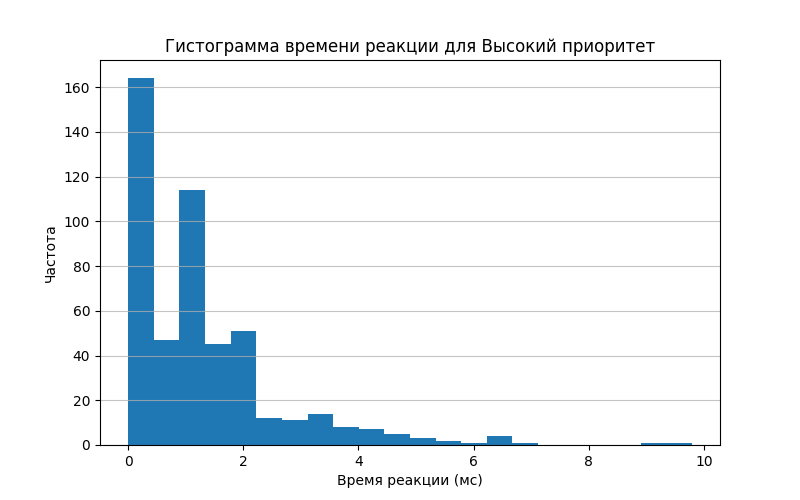


Рисунок 2 — Гистограмма времени реакции на событие при высоком приоритете.

1. Низкий приоритет (значение приоритета 19) (см. рис. Рисунок 3):

Среднее время отклика = 1.31712 мс.

Максимальное время отклика = 23.15756 мс.

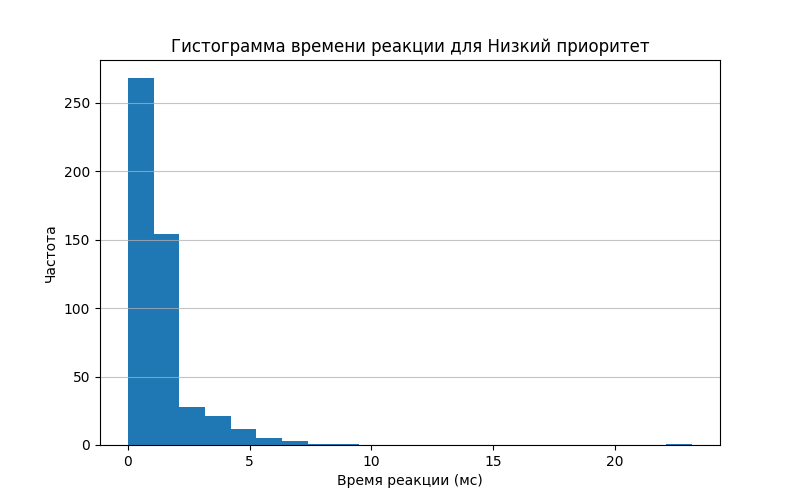


Рисунок 3 — Гистограмма времени реакции на событие при низком приоритете.

1. Дисциплина диспетчеризации SCHED\_BATCH (см. рис. Рисунок 4):

Среднее время отклика = 1.63377 мс.

Максимальное время отклика = 20.26886 мс.

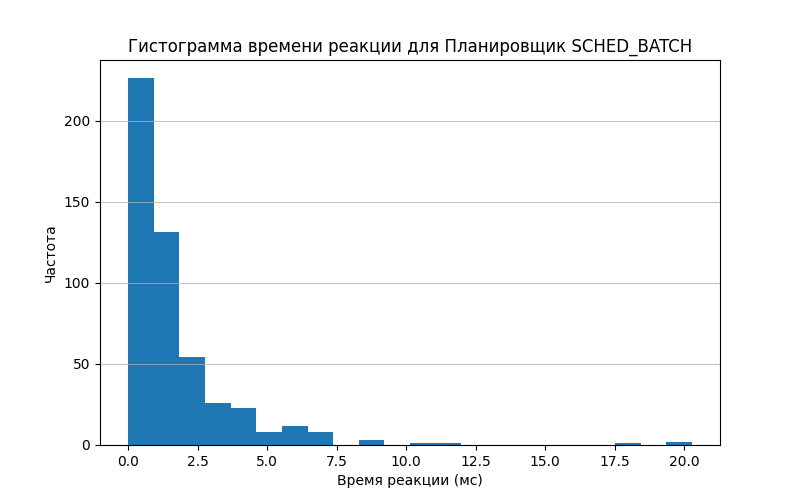


Рисунок 4 — Гистограмма времени реакции на событие при SCHED\_BATCH.

1. Дисциплина диспетчеризации SCHED\_FIFO (см. рис. Рисунок 5):

Среднее время отклика = 0.95466 мс.

Максимальное время отклика = 16.02450 мс.



Рисунок 5 — Гистограмма времени реакции на событие при SCHED\_FIFO.

1. Дисциплина диспетчеризации SCHED\_RR (см. рис. Рисунок 6):

Среднее время отклика = 1.87247 мс.

Максимальное время отклика = 18.74743 мс.

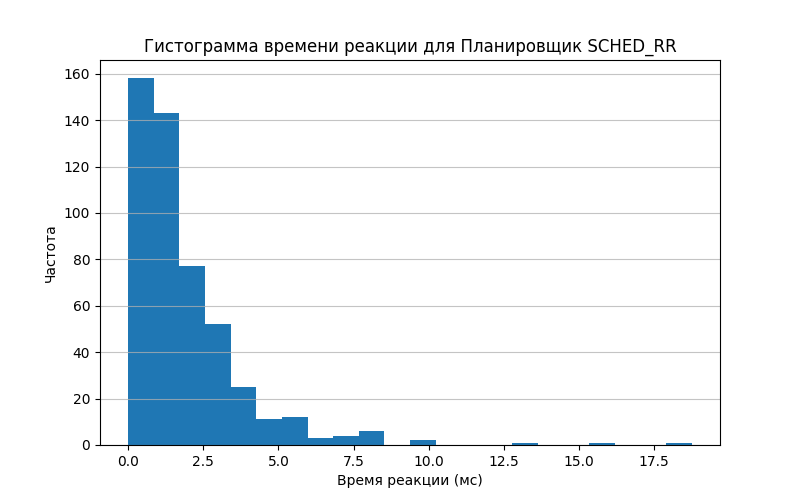


Рисунок 6 — Гистограмма времени реакции на событие при SCHED\_RR.

1. Дисциплина диспетчеризации SCHED\_IDLE (см. рис. Рисунок 7):

Среднее время отклика = 1.20158 мс.

Максимальное время отклика = 9.36210 мс.

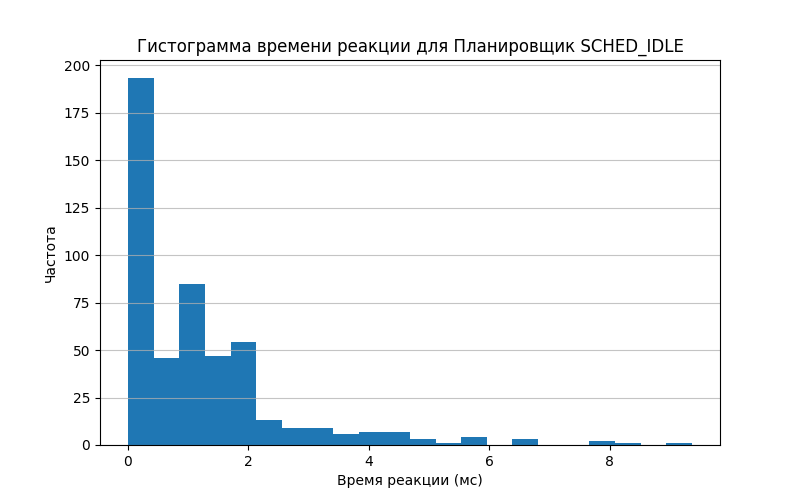


Рисунок 7 — Гистограмма времени реакции на событие при SCHED\_IDLE.

1. Дисциплина диспетчеризации SCHED\_NORMAL/OTHER (см. рис. Рисунок 8):

Дисциплины диспетчеризации SCHED\_NORMAL и SCHED\_OTHER не имеют различий и относятся к обычному планировщику CFS. Поэтому проведен тест только для SCHED\_OTHER.

Среднее время отклика = 2.37530 мс.

Максимальное время отклика = 69.36455 мс.

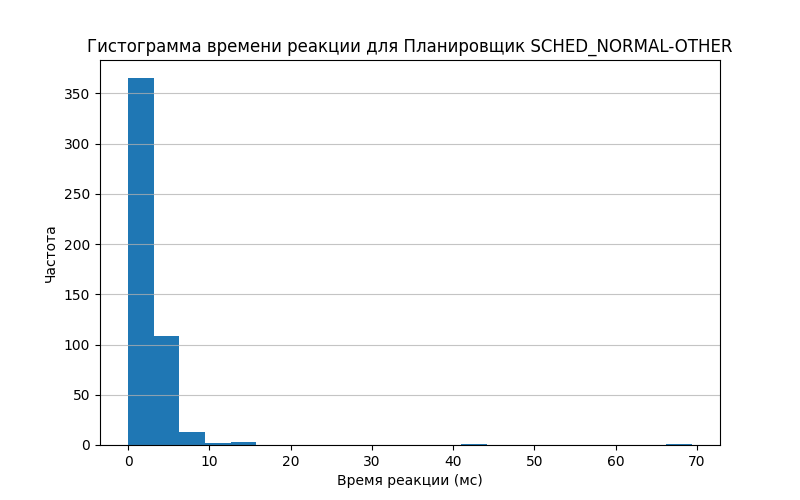


Рисунок 8 — Гистограмма времени реакции на событие при SCHED\_NORMAL/OTHER.

1. Под нагрузкой HDD (см. рис. Рисунок 9):

Для симуляции нагрузки на жёсткий диск использована утилита dd. Она записывает в файл test.file 1000 блоков по 10 мегабайт, заполненных нулями.

Среднее время отклика = 1.42230 мс.

Максимальное время отклика = 35.05833 мс.

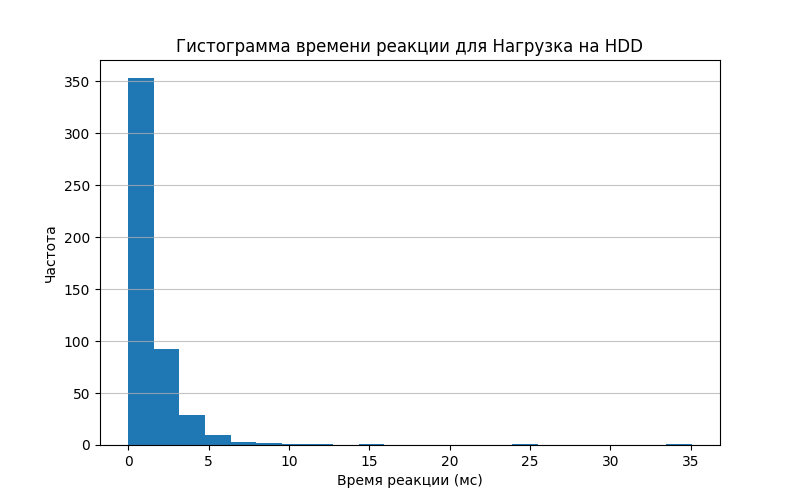


Рисунок 9 — Гистограмма времени реакции на событие при нагрузке HDD.

1. Под нагрузкой сети (см. рис. Рисунок 10)

Для симуляции нагрузки на сеть использована утилита wget. Во время выполнения программы качался образ Ubuntu с официального сайта.

Среднее время отклика = 2.17121 мс.

Максимальное время отклика = 13.16306 мс.

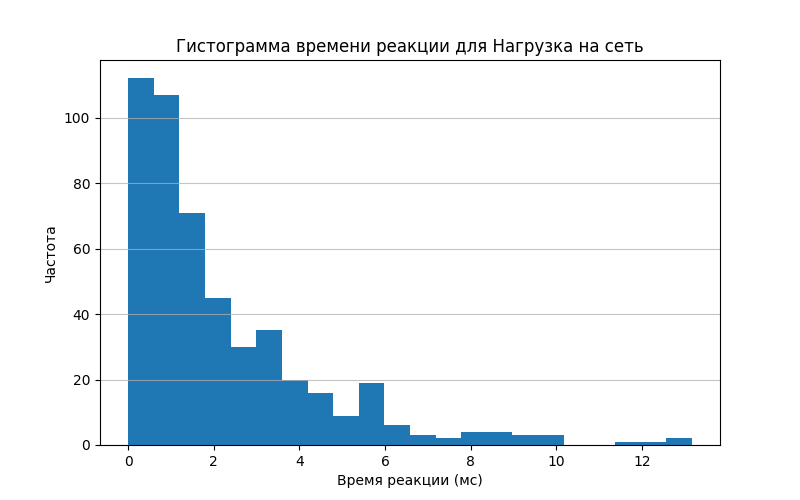


Рисунок 10 — Гистограмма времени реакции на событие при нагрузке сети.

Вывод

В лабораторной работе №2 разработан символьный драйвер для оценки нагрузки сети. Проведены тесты на ОС Ubuntu 22.04. Повышенный приоритет улучшает максимальное время реакции. Планировщик SCHED\_FIFO дает наилучшие результаты по среднему времени реакции. SCHED\_IDLE также показывает хорошие результаты, но может быть не всегда подходящим для приложений, которым нужно быстро реагировать. Нагрузка на диск и сеть увеличивает время реакции. Особенно сильно на это заметно при нагрузке на диск. Также имеют место значительные различия между минимальным и максимальным временем реакции, что говорит о нестабильности системы и влиянии многих факторов на нёё.

Приложение А. Код драйвера

Код программы символьного драйвера написан на языке C.

#include <linux/cdev.h>

#include <linux/device.h>

#include <linux/fs.h>

#include <linux/ioctl.h>

#include <linux/kernel.h>

#include <linux/ktime.h>

#include <linux/module.h>

#include <linux/sched.h>

#include <linux/slab.h>

#include <linux/timer.h>

#include <linux/uaccess.h>

#define DEVICE\_NAME "mytest"

#define INITIAL\_BUFFER\_SIZE 100

// ioctl command

#define MYTEST\_GET\_REACTIONS \_IOR('m', 0, ktime\_t \*)

#define MYTEST\_GET\_COUNT \_IOR('m', 1, int \*)

static struct timer\_list my\_timer;

static int frequency = 1;

static ktime\_t event\_time;

static ktime\_t \*response\_time = NULL;

static int buffer\_size = INITIAL\_BUFFER\_SIZE;

static int index = 0;

static bool event\_pending = false;

static bool reaction\_pending = false;

static dev\_t dev\_num;

static struct cdev my\_cdev;

static struct class \*dev\_class;

module\_param(frequency, int, 0);

static int device\_open(struct inode \*inode, struct file \*file) {

// Reset response buffer and index

if (response\_time != NULL) {

kfree(response\_time);

response\_time = NULL;

}

index = 0;

buffer\_size = INITIAL\_BUFFER\_SIZE;

response\_time = kmalloc(sizeof(ktime\_t) \* buffer\_size, GFP\_KERNEL);

if (!response\_time) {

printk(KERN\_ERR "Failed to allocate buffer in open\n");

return -ENOMEM;

}

return 0;

}

static ssize\_t device\_read(struct file \*file, char \_\_user \*buffer, size\_t len,

loff\_t \*offset) {

if (!event\_pending) return 0;

reaction\_pending = true;

return 1;

}

static ssize\_t device\_write(struct file \*file, const char \_\_user \*buffer, size\_t len, loff\_t \*offset) {

ktime\_t reaction\_time;

if (!reaction\_pending) return -EINVAL;

ktime\_t current\_time = ktime\_get();

reaction\_time = ktime\_sub(current\_time, event\_time);

if (index >= buffer\_size) {

buffer\_size \*= 2;

response\_time =

krealloc(response\_time, sizeof(ktime\_t) \* buffer\_size, GFP\_KERNEL);

if (!response\_time) {

return -ENOMEM;

}

}

response\_time[index++] = reaction\_time;

printk(KERN\_INFO "Reaction Time: %lld ns\n", ktime\_to\_ns(reaction\_time));

event\_pending = false;

reaction\_pending = false;

return len;

}

void timer\_callback(struct timer\_list \*timer) {

if (event\_pending) {

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(1000 / frequency));

return;

}

event\_time = ktime\_get();

event\_pending = true;

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(1000 / frequency));

printk(KERN\_INFO "External Event Triggered\n");

}

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

switch (cmd) {

case MYTEST\_GET\_REACTIONS:

if (response\_time == NULL) {

return -EFAULT;

}

if (copy\_to\_user((ktime\_t \*)arg, response\_time, sizeof(ktime\_t) \* index))

return -EFAULT;

printk(KERN\_INFO "Returning %d reaction times via ioctl\n", index);

break;

case MYTEST\_GET\_COUNT:

if (copy\_to\_user((int \*)arg, &index, sizeof(int))) return -EFAULT;

printk(KERN\_INFO "Returning %d as reaction count via ioctl\n", index);

break;

default:

return -ENOTTY;

}

return 0;

}

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = device\_open,

.read = device\_read,

.write = device\_write,

.unlocked\_ioctl = device\_ioctl,

};

static int \_\_init mytest\_init(void) {

int result;

result = alloc\_chrdev\_region(&dev\_num, 0, 1, DEVICE\_NAME);

if (result < 0) {

printk(KERN\_ALERT "Failed to allocate device number\n");

return result;

}

cdev\_init(&my\_cdev, &fops);

my\_cdev.owner = THIS\_MODULE;

result = cdev\_add(&my\_cdev, dev\_num, 1);

if (result < 0) {

printk(KERN\_ALERT "Failed to add character device\n");

unregister\_chrdev\_region(dev\_num, 1);

return result;

}

dev\_class = class\_create(DEVICE\_NAME);

if (IS\_ERR(dev\_class)) {

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

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

return PTR\_ERR(dev\_class);

}

if (IS\_ERR(device\_create(dev\_class, NULL, dev\_num, NULL, DEVICE\_NAME))) {

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

class\_destroy(dev\_class);

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

return PTR\_ERR(device\_create(dev\_class, NULL, dev\_num, NULL, DEVICE\_NAME));

}

response\_time = kmalloc(sizeof(ktime\_t) \* buffer\_size, GFP\_KERNEL);

if (!response\_time) {

device\_destroy(dev\_class, dev\_num);

class\_destroy(dev\_class);

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

return -ENOMEM;

}

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

mod\_timer(&my\_timer, jiffies + msecs\_to\_jiffies(1000 / frequency));

printk(KERN\_INFO "mytest Driver Initialized\n");

return 0;

}

static void \_\_exit mytest\_exit(void) {

del\_timer(&my\_timer);

if (response\_time != NULL) kfree(response\_time);

device\_destroy(dev\_class, dev\_num);

class\_destroy(dev\_class);

cdev\_del(&my\_cdev);

unregister\_chrdev\_region(dev\_num, 1);

printk(KERN\_INFO "mytest Driver Exited\n");

}

module\_init(mytest\_init);

module\_exit(mytest\_exit);

MODULE\_LICENSE("GPL");

Приложение Б. Код программы

Код программы приложения написан на языке C.

#include <fcntl.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ioctl.h>

#include <time.h>

#include <unistd.h>

#define DEVICE\_FILE "/dev/mytest"

#define NUM\_OF\_REACTIONS 500

// ioctl command

#define MYTEST\_GET\_REACTIONS \_IOR('m', 0, time\_t \*)

#define MYTEST\_GET\_COUNT \_IOR('m', 1, int \*)

int main() {

int fd, i;

char buffer[1024];

ssize\_t bytes\_read;

long long \*reaction\_times;

int num\_times = 0;

int reaction\_count;

fd = open(DEVICE\_FILE, O\_RDWR);

if (fd < 0) {

perror("Failed to open device");

return 1;

}

int try\_count = 0;

i = 0;

while (i < NUM\_OF\_REACTIONS) {

bytes\_read = read(fd, buffer, 1);

if (bytes\_read <= 0) {

try\_count++;

if (try\_count % 100 == 0) {

// printf("No event available\n");

try\_count = 0;

}

usleep(1000);

continue;

}

i++;

// send reaction

int dummy\_id = 0;

if (write(fd, &dummy\_id, sizeof(int)) == -1) {

perror("Error while sending response");

}

// printf("Event %d received\n", i);

}

// Get number of reaction times via ioctl

if (ioctl(fd, MYTEST\_GET\_COUNT, &reaction\_count) == -1) {

perror("ioctl MYTEST\_GET\_COUNT failed");

close(fd);

return 1;

}

printf("Number of reactions is: %d\n", reaction\_count);

// Allocate memory for reactions

reaction\_times = (long long \*)malloc(sizeof(long long) \* reaction\_count);

if (reaction\_times == NULL) {

perror("Error while allocating memory");

close(fd);

return 1;

}

// Get reaction times via ioctl

if (ioctl(fd, MYTEST\_GET\_REACTIONS, reaction\_times) == -1) {

perror("ioctl MYTEST\_GET\_REACTIONS failed");

free(reaction\_times);

close(fd);

return 1;

}

printf("Recorded %d reaction times:\n", reaction\_count);

for (i = 0; i < reaction\_count; ++i) {

printf("Reaction time %d: %lld ns\n", i, reaction\_times[i]);

}

free(reaction\_times);

close(fd);

return 0;

}