Федеральное государственное автономное
образовательное учреждение высшего образования
«Научно-образовательная корпорация ИТМО»

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Программная инженерия

Отчёт по лабораторной работе №2

По дисциплине «Системы ввода-вывода» (семестр 6)

Студент:
Дениченко Александр P3312
Разинкин Александр P3307
Практик:
Табунщик Сергей Михайлович

Санкт-Петербург
2025 г.

Цель

Познакомится с основами разработки драйверов устройств с использованием операционной системы на примере создания драйверов символьных устройств под операционную систему Linux.

1 Задачи

Написать драйвер символьного устройства, удовлетворяющий требованиям:

- должен создавать символьное устройство /dev/varN, где N – это номер варианта
- должен обрабатывать операции записи и чтения в соответствии с вариантом задания

2 Вариант

При записи текста в файл символьного устройства должно запоминаться количество пробелов во введенном тексте. Последовательность полученных результатов с момента загрузки модуля ядра должна выводиться при чтении файла.

3 Выполнение

Функция my_read является обработчиком системного вызова read() для данного символьного устройства:

Листинг 1: my_read

```
1  static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)
2  {
3      printk(KERN_INFO "Driver: read()\n");
4
5      result *curr_res;
6      int ptr = 0;
7
8      curr_res = first_result;
9      while (curr_res) {
10         ptr += sprintf(lbuf + ptr, "%ld ", curr_res->spaces);
11         curr_res = curr_res->next;
12     }
13
14     lbuf[ptr++] = '\n';
15     lbuf[ptr++] = '\0';
16
17     size_t count = strlen(lbuf);
18
19     if (*off > 0 || len < count) {
20         return 0;
21     }
22
23     if (copy_to_user(buf, lbuf, count) != 0) {
24         return -EFAULT;
25     }
26
27     *off = count;
28
29     return count;
30 }
```

Проходит по связному списку результатов, формирует строку, содержащую все значения, разделенные пробелами. Копирует данные в пользовательское пространство. Обновляет смещение в файле и возвращает количество скопированных байт.

Функция my_write является обработчиком системного вызова write() для данного символьного устройства:

Листинг 2: my_write

```
static ssize_t my_write(struct file *f, const char __user *buf,  size_t len, loff_t *off)
{
  printk(KERN_INFO "Driver: write()\n");

  if (len > BUF_SIZE) {
    return 0;
  }

  if (copy_from_user(lbuf, buf, len) != 0) {
    return -EFAULT;
  }

  result *res = (result *) kmalloc(sizeof(result), GFP_KERNEL);
  if (!res) {
    printk(KERN_ERR "Can not allocate memory for driver data.\n");
    return 0;
  }

  size_t size = strlen(lbuf);
  size_t spaces = 0;
  size_t i = 0;
  while (i != size) {
    if (lbuf[i] == ' ') {
      spaces++;
    }
    i++;
  }

  res->spaces = spaces;

  if (!last_result) {
    first_result = res;
    last_result = res;
  } else  {
    last_result->next = res;
    last_result = res;
  }
  res->next = NULL;

  return len;
}
```

Проверяет, не превышает ли размер данных размер буфера. Копирует данные из пользовательского пространства в буфер ядра. Подсчитывает количество пробелов в полученной строке. Сохраняет результат в конец связного списка результатов и возвращает кол-во записанных байт.

4 Полный код

Листинг 3: ch_drv.c

```c
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/slab.h>

#define BUF_SIZE 256

static dev_t first;
static struct cdev c_dev;
static struct class *cl;

static char lbuf[BUF_SIZE];

typedef struct result {
  size_t spaces;
  struct result *next;
} result;

static result *first_result, *last_result;

static int my_open(struct inode *i, struct file *f)
{
  printk(KERN_INFO "Driver: open()\n");
  return 0;
}

static int my_close(struct inode *i, struct file *f)
{
  printk(KERN_INFO "Driver: close()\n");
  return 0;
}

static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
  printk(KERN_INFO "Driver: read()\n");

  result *curr_res;
  int ptr = 0;

  curr_res = first_result;
  while (curr_res) {
    ptr += sprintf(lbuf + ptr, "%ld ", curr_res->spaces);
    curr_res = curr_res->next;
  }

  lbuf[ptr++] = '\n';
  lbuf[ptr++] = '\0';

  size_t count = strlen(lbuf);

  if (*off > 0 || len < count) {
```

```
57      return 0;
58    }
59
60    if (copy_to_user(buf, lbuf, count) != 0) {
61      return -EFAULT;
62    }
63
64    *off = count;
65
66    return count;
67  }
68
69  static ssize_t my_write(struct file *f, const char __user *buf,  size_t len, loff_t *off)
70  {
71    printk(KERN_INFO "Driver: write()\n");
72
73    if (len > BUF_SIZE) {
74      return 0;
75    }
76
77    if (copy_from_user(lbuf, buf, len) != 0) {
78      return -EFAULT;
79    }
80
81    result *res = (result *) kmalloc(sizeof(result), GFP_KERNEL);
82    if (!res) {
83      printk(KERN_ERR "Can not allocate memory for driver data.\n");
84      return 0;
85    }
86
87    size_t size = strlen(lbuf);
88    size_t spaces = 0;
89    size_t i = 0;
90    while (i != size) {
91      if (lbuf[i] == ' ') {
92        spaces++;
93      }
94      i++;
95    }
96
97    res->spaces = spaces;
98
99    if (!last_result) {
100     first_result = res;
101     last_result = res;
102   } else {
103     last_result->next = res;
104     last_result = res;
105   }
106   res->next = NULL;
107
108   return len;
109 }
110
111 static struct file_operations mychdev_fops =
112 {
```

```
113     .owner = THIS_MODULE,
114     .open = my_open,
115     .release = my_close,
116     .read = my_read,
117     .write = my_write
118 };
119
120 static int __init ch_drv_init(void)
121 {
122     printk(KERN_INFO "Hello!\n");
123     if (alloc_chrdev_region(&first, 0, 1, "ch_dev") < 0){
124             return -1;
125     }
126     if ((cl = class_create(THIS_MODULE, "chardrv")) == NULL){
127             unregister_chrdev_region(first, 1);
128             return -1;
129     }
130     if (device_create(cl, NULL, first, NULL, "mychdev") == NULL){
131             class_destroy(cl);
132             unregister_chrdev_region(first, 1);
133             return -1;
134     }
135     cdev_init(&c_dev, &mychdev_fops);
136     if (cdev_add(&c_dev, first, 1) == -1){
137             device_destroy(cl, first);
138             class_destroy(cl);
139             unregister_chrdev_region(first, 1);
140             return -1;
141     }
142     return 0;
143 }
144
145 static void __exit ch_drv_exit(void)
146 {
147     cdev_del(&c_dev);
148     device_destroy(cl, first);
149     class_destroy(cl);
150     unregister_chrdev_region(first, 1);
151     printk(KERN_INFO "Bye!!!\n");
152 }
153
154 module_init(ch_drv_init);
155 module_exit(ch_drv_exit);
156
157 MODULE_LICENSE("GPL");
158 MODULE_AUTHOR("Author");
159 MODULE_DESCRIPTION("The first kernel module");
```