Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Драйвер символьного дисплея»

Студент ИУ7-72Б _____ Карпова Е. О.
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ Рязанова Н. Ю.
(Подпись, дата) (И. О. Фамилия)

2024 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ-7
_____ И. В. Рудаков
«16» сентября 2023 г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине

Операционные системы

по теме

«Драйвер символьного дисплея»

Студент группы **ИУ7-72Б**

Карпова Екатерина Олеговна

График выполнения работы: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание

Разработать программный комплекс, состоящий из программы уровня пользователя, получающей информацию о конкретном процессе, и драйвера символьного дисплея, обеспечивающего вывод информации, заданной приложением.

Оформление курсовой работы:

Расчетно-пояснительная записка на **25-40** листах формата А4.

Дата выдачи задания «16» сентября 2023 г.

Руководитель курсовой работы _____ **Рязанова Н. Ю.**
(Подпись, дата) (Фамилия И. О.)

Студент _____ **Карпова Е. О.**
(Подпись, дата) (Фамилия И. О.)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Драйверы устройств играют особую роль в ядре Linux [1]. Каждый драйвер реализует определённый программный интерфейс взаимодействия с внешним устройством, скрывая от пользователя подробности работы устройства. Сопоставление вызовов, представленных в интерфейсе, с операциями, специфичными для устройства — основная задача драйвера устройства в Linux-системах. Реализация драйвера как модуля ядра Linux позволяет расширять функциональность ядра по работе с внешними устройствами во время работы системы. Часто вместе с драйвером предоставляется клиентская библиотека, реализующая возможности, не являющиеся частью интерфейса самого драйвера.

Некоторые драйверы также работают с дополнительными наборами функций ядра для данного типа устройств [1]. Например, с интерфейсами USB, I2C, UART, SPI и другими. Наиболее распространёнными интерфейсами для работы с символьными дисплеями являются I2C, SPI и UART.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием целью данной работы является разработка программного комплекса, состоящего из программы уровня пользователя, получающей информацию о конкретном процессе, и драйвера символьного дисплея, обеспечивающего вывод информации, заданной приложением.

В ходе работы необходимо решить следующие задачи:

1) Провести сравнительный анализ способов реализации драйверов устройств.

2) Сделать выбор способа реализации драйвера символьного дисплея.

3) Описать способ получения информации о процессе по его идентификатору из пространства пользователя.

4) Провести сравнительный анализ существующих интерфейсов взаимодействия с символьным жидкокристаллическим дисплеем.

5) Сделать выбор интерфейса взаимодействия с символьным жидкокристаллическим дисплеем в соответствии с поставленной целью.

6) Сделать выбор системных вызовов, поддерживаемых драйвером, в соответствии с поставленной целью.

7) Разработать алгоритмы и структуру ПО.

8) Реализовать драйвер символьного дисплея для вывода информации о конкретном процессе как загружаемый модуль ядра Linux.

9) Реализовать программу уровня пользователя для взаимодействия с символьным дисплеем.

1.2 Способы реализации драйверов устройств

Драйвер устройства — программа, управляющая работой внешнего устройства со стороны системы. Существуют драйверы уровня пользователя и драйверы в формате загружаемого модуля ядра.

1) Драйверы уровня пользователя предоставляют интерфейс между приложениями и драйверами уровня ядра. Например, драйверы принтеров.

2) Драйверы уровня ядра выполняются на уровне ядра, используя его структуры и функции. Обычно такие драйверы разделяются на подуровни. В основном верхние уровни получают информацию от приложений, обрабатывают ее и передают драйверам нижнего уровня. К этой категории относятся, например, драйверы файловых систем.

1.3 Выбор способа реализации драйвера символьного дисплея

В UNIX-системах для обращения к драйверу, реализованному как загружаемый модуль ядра, используются специальные файлы устройств. Специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре. Каждому устройству соответствует хотя бы один специальный файл. Обычно они лежат в каталоге `/dev` корневой файловой системы. За счет этого обеспечивается унифицированный доступ к периферийным устройствам — с ними можно взаимодействовать, как с обычными файлами: открывать, закрывать, читать, писать. Поэтому в данной работе было принято решение реализовать драйвер устройства как загружаемый модуль ядра.

1.4 Способ получения информации о процессе по его PID из пространства пользователя

Виртуальная файловая система `proc` не является монтируемой файловой системой поэтому и называется виртуальной. Её корневым каталогом является каталог `/proc`. Основная задача файловой системы `proc` — предоставление информации процессам о занимаемых ими ресурсах, что совпадает с поставленной в работе целью.

Каждый процесс в файловой системе `proc` имеет поддиректорию: `/proc/<PID>`. В данной директории находятся файлы и другие вложеные директории, содержащие информацию о данном процессе.

В данной работе будут использованы:

1) cmdline — файл, содержащий командную строку запуска процесса;

2) fd — директория, содержащая ссылки на файлы, открытые процессом;

3) tasks — директория, содержащая поддиректории потоков;

4) stat, statm — файлы, содержащие информацию о состоянии процесса;

5) comm — файл, содержащий имя исполняемого файла процесса.

1.5 Интерфейсы взаимодействия с внешними устройствами

1.5.1 Последовательная и параллельная связь

Для взаимодействия между устройствами должен быть определен протокол связи. Выделяется 2 основных вида связи:

1) Последовательная.

2) Параллельная.

Последовательная связь может быть реализована с использованием меньшего количества проводов, однако требует настройки механизма синхронизации [2] и обеспечивает меньшую скорость передачи по сравнению с параллельной [3]. Также, последовательная связь является наиболее широко используемой коммуникационной методологией. Исходя из данных особенностей, был выбран последовательный вид связи, и далее будут рассмотрены поддерживающие его интерфейсы.

1.5.2 SPI

Интерфейс SPI представляет собой последовательный четырехканальный интерфейс синхронной передачи данных [4]. В SPI передача данных синхронизирована с тактовым сигналом основного устройства. Периферийные устройства синхронизируют получение битовой последовательности с тактовым сигналом. К одному интерфейсу ведущего устройства может подключаться несколько периферийных устройств. Ведущее устройство обрабатывает все данные и выбирает ведомое для передачи. Периферия, не выбранная ведущим устройством, не принимает участия в передаче по SPI. Шина SPI способна передавать информацию сразу в двух направлениях, стандартная скорость обмена данными до 20 Мбит/с.

Преимущества интерфейса SPI:

1) Надежность синхронного типа связи.

2) Возможность подключения нескольких периферийных устройств.

3) Скорость передачи данных выше, чем у UART и I2C.

Недостатки интерфейса SPI:

1) Требуется минимум 4 линии связи, количество каналов растёт при увеличении числа подключаемых устройств.

2) Только ведущий контролирует весь процесс коммуникации, нет прямой связи между ведомыми.

1.5.3 UART

Интерфейс UART обепечивает последовательную асинхронную связь. UART работает путем перевода между параллельной связью и последовательной связью [3]. Поддерживает дуплексный режим. Наиболее распространённые скорости передачи — до 112 Кбит/с.

Необходимо только два провода. Поскольку связь асинхронная, оба взаимодействующих устройства должны использовать свои независимые внутренние системы тактирования для функционирования. Тем не менее, существует термин «скорость передачи», который помогает этим устройствам оставаться в режиме синхронизации, фиксируя скорость обмена данными. Скорость передачи данных в бодах равно число бит данных, передаваемое в секунду, поэтому оба устройства должны работать с одинаковой скоростью передачи, чтобы поддерживать его надлежащее функционирование. Время согласовывается заранее между обеими единицами, и к каждому пакету данных добавляются специальные биты — слово. UART используют эти биты для синхронизации друг с другом. Интерфейс UART имеет большое ограничение, связанное с тем, что только два устройства могут обмениваться данными с помощью этого протокола одновременно.

Преимущества интерфейса UART:

1) Обеспечивается наличие различных скоростных режимов передачи, что делает протокол практически универсальным.

2) Является одной из самых простых форм последовательной связи.

Недостатки интерфейса UART:

1) Одновременно могут быть подключены только два устройства.

2) Требуется передача дополнительных битов для согласования времени передачи.

3) Скорость передачи данных ниже, чем у I2C и SPI.

1.5.4 I2C

Интерфейс I2C обепечивает последовательную синхронную связь [5]. Данные передаются по двум линиям: данных и тактирования. Обе линии поддерживают дуплексный режим. Устройства, участвующие в транзакциях, разделяются на ведущее устройство и ведомые им устройства. Ведущее определяет начало и конец обмена данными, синхронизирует транзакции и опрашивает ведомые устройства. Максимальное количество периферийных устройств для подключения к шине — 127. Данные по шине I2C могут передаваться со скоростью до 100 – 200 Кбит/с в стандартном режиме, и до 400 Кбит/с в быстром режиме.

Каждая посылка данных состоит из 8 бит данных, формируемых передатчиком. Приемник во время девятого такта формирует бит подтверждения, по которому передатчик убеждается, что передача прошла успешно. После передачи бита подтверждения ведомое устройство может начать следующую передачу.

Преимущества интерфейса I2C:

1) Необходимо 2 линии связи, независимо от количества подключенных устройств.

2) Возможность подключения нескольких ведущих устройств.

3) Протокол I2C является более стандартизованным, поэтому, пользователь более защищен от проблем несовместимости выбранных компонентов.

4) Позволяет взаимодействовать на одной шине устройствам с различным быстродействием интерфейсов.

Недостатки интерфейса I2C:

1) Скорость передачи данных ниже, чем у SPI, из-за большого количества операций с кадрами данных.

1.6 Выбор интерфейса взаимодействия с символьным жидкокристаллическим дисплеем

В таблице 1.1 приведен сравнительный анализ рассмотренных интерфейсов взаимодействия с внешними устройствами.

Таблица 1.1 – Сравнительный анализ рассмотренных интерфейсов взаимодействия с внешними устройствами

Интерфейсы взаимодействия с внешними устройствами	Критерии сравнения			
	Скорость передачи данных (до Кбит/с)	Кол-во каналов (мин.)	Надежность передачи	Требует передачи доп. данных
UART	112	2	–	+
SPI	20 480	4	+	–
I2C	200	2	+	–

В результате сравнения было принято решение использовать интерфейс I2C, так как он является более надежным, не требует большого числа каналов и передачи дополнительных данных. Скорость передачи не являлась ключевым критерием в соответствии с целью работы, поэтому, несмотря на то, что SPI превосходит I2C по этому критерию, он, как требующий наличия большего числа каналов, не был выбран.

Вывод

Было принято решение реализовать драйвер символьного дисплея для вывода информации о процессе как модуль ядра Linux. Также, был проведен сравнительный анализ интерфейсов взаимодействия с внешними устройствами, в результате которого был выбран интерфейс I2C.

2 Конструкторский раздел

2.1 Схема программного обеспечения

На рисунке 2.1 представлена IDEF0-диаграмма нулевого уровня разрабатываемого программно-аппаратного комплекса.

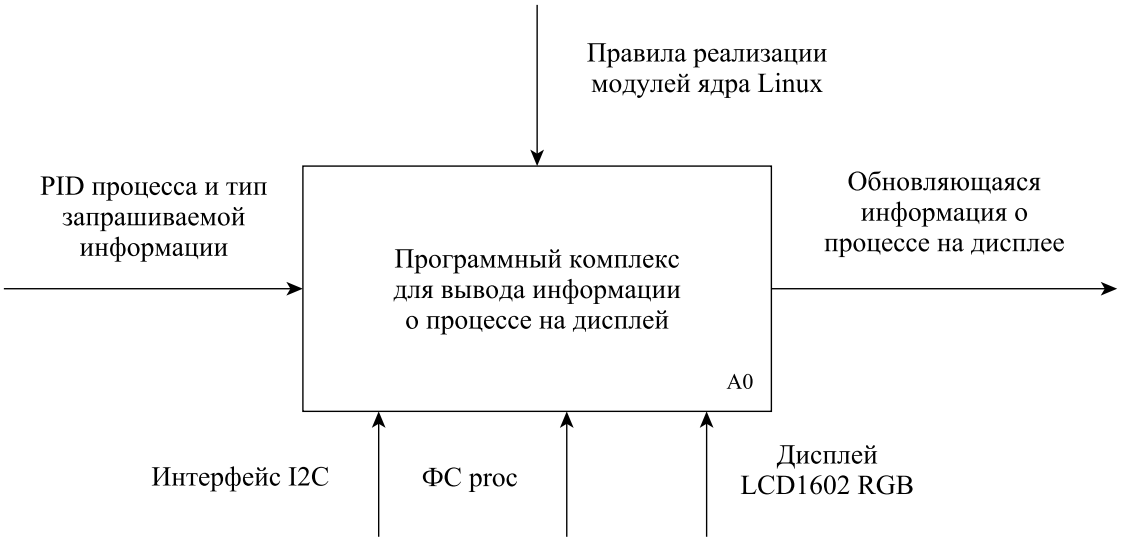

Рисунок 2.1 – IDEF0-диаграмма нулевого уровня разрабатываемого программно-аппаратного комплекса

На рисунке 2.2 представлена IDEF0-диаграмма первого уровня разрабатываемого программно-аппаратного комплекса.

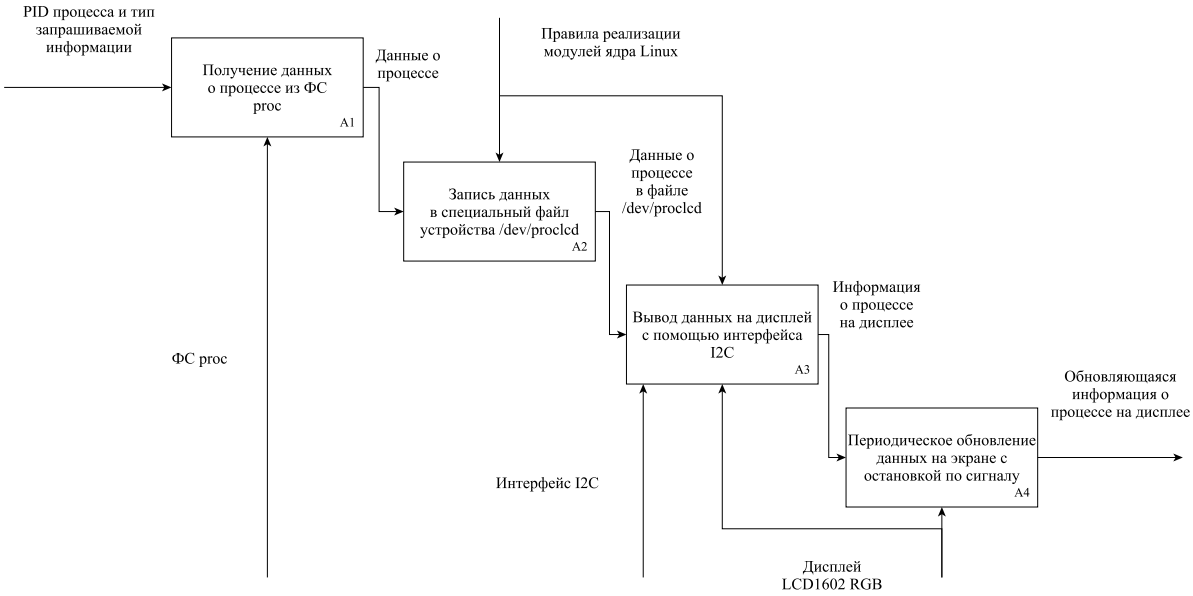

Рисунок 2.2 – IDEF0-диаграмма первого уровня разрабатываемого программно-аппаратного комплекса

2.2 Схема алгоритма инициализации модуля ядра Linux — драйвера символьного дисплея

На рисунке 2.3 представлена схема алгоритма инициализации модуля ядра Linux — драйвера символьного дисплея.

Рисунок 2.3 – Схема алгоритма инициализации модуля ядра Linux — драйвера символьного дисплея

2.3 Схема алгоритма вывода данных на символьный дисплей

На рисунке 2.4 представлена схема алгоритма вывода данных на символьный дисплей.

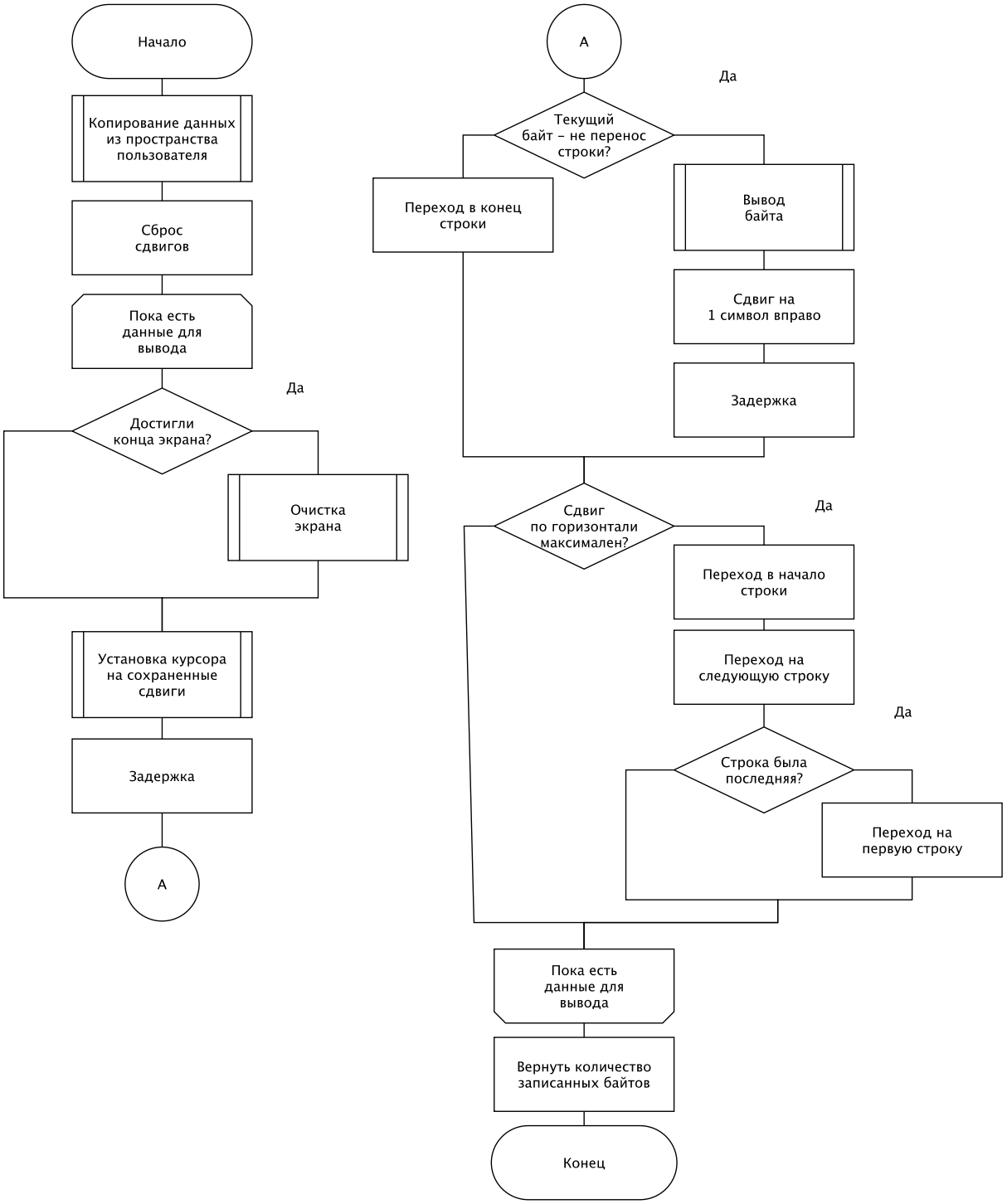

Рисунок 2.4 – Схема алгоритма вывода данных на символьный дисплей

2.4 Схема алгоритма чтения данных из устройства

На рисунке 2.5 представлена схема алгоритма чтения данных из устройства.

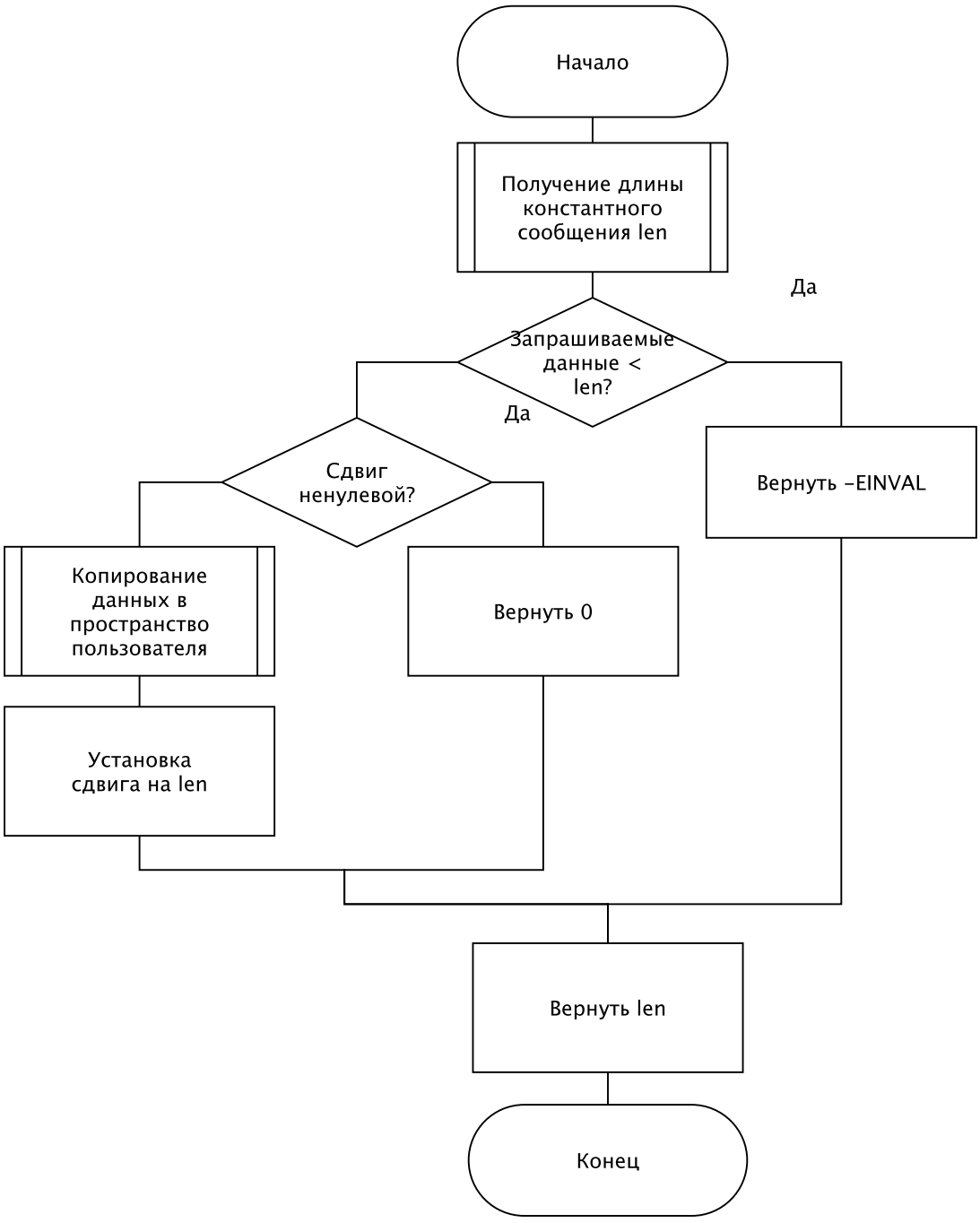

Рисунок 2.5 – Схема алгоритма чтения данных из устройства

2.5 Схема алгоритма записи данных в шину через интерфейс I2C

На рисунке 2.6 представлена схема алгоритма записи данных в шину через интерфейс I2C.

Рисунок 2.6 – Схема алгоритма записи данных в шину через интерфейс I2C

2.6 Схема алгоритма работы клиентского приложения

На рисунке 2.7 представлена схема алгоритма работы клиентского приложения.

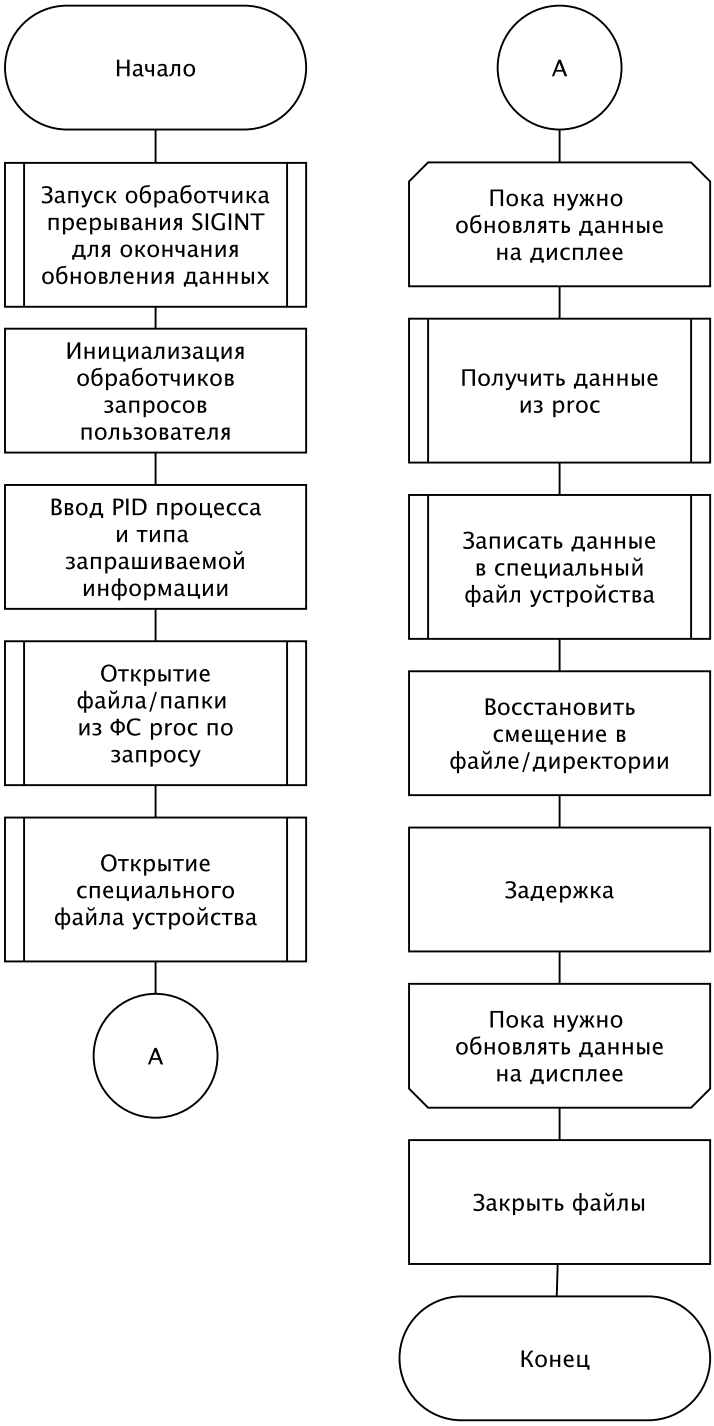

Рисунок 2.7 – Схема алгоритма работы клиентского приложения

2.7 Структуры ядра

2.7.1 struct file_operations

struct file_operations — структура, содержащая либо стандартные операции на файлах для конкретной файловой системы, либо зарегистрированные разработчиком.

Листинг 2.1 – struct file_operations

```
1  struct file_operations {
2    struct module *owner;
3    loff_t (*llseek) (struct file *, loff_t, int);
4    ssize_t (*read) (struct file *, char __user *, size_t, loff_t
       *);
5    ssize_t (*write) (struct file *, const char __user *, size_t,
       loff_t *);
6    ...
7    int (*open) (struct inode *, struct file *);
8    int (*flush) (struct file *, fl_owner_t id);
9    int (*release) (struct inode *, struct file *);
10   ...
11 } __randomize_layout;
```

2.7.2 struct device

struct device — базовая структура устройства, редко используется в чистом виде. parent — устройство, к которому подсоединено данное устройство, init_name — имя устройства, type — тип устройства, bus — тип шины.

Листинг 2.2 – struct device

```
1  struct device {
2    ...
3    struct device   *parent;
4    ...
5    const char     *init_name; /* initial name of the device */
6    const struct device_type *type;
7
8    struct bus_type *bus;    /* type of bus device is on */
9    struct device_driver *driver; /* which driver has allocated
       this
10                 device */
11   void     *platform_data; /* Platform specific data, device
```

```
12                core doesn't touch it */
13      void     *driver_data; /* Driver data, set and get with
14                dev_set_drvdata/dev_get_drvdata */
15      struct mutex     mutex;  /* mutex to synchronize calls to
16               * its driver.
17               */
18      ...
19  };
```

2.7.3 struct miscdevice

struct miscdevice — структура, описывающая устройство. Используется при реализации небольших драйверов и является упрощённой по сравнению с struct cdev [6]. Система хранит все старшие номера устройств в статической таблице, поэтому выделение старших номеров под простые драйверы считается излишним. Устройства, зарегистрированные, как miscdevice, получают старший номер равный 10. Также, при использовании miscdevice, специальный файл устройства создается сразу при регистрации устройства, а при выборе cdev его нужно создавать отдельно.

Листинг 2.3 – struct miscdevice

```
1  struct miscdevice  {
2    int minor;
3    const char *name;
4    const struct file_operations *fops;
5    struct list_head list;
6    struct device *parent;
7    struct device *this_device;
8    const struct attribute_group **groups;
9    const char *nodename;
10   umode_t mode;
11 };
```

2.7.4 struct file

struct file — структура, которая описывает открытый файл.

Листинг 2.4 – struct file

```
1  struct file {
2    ...
3    struct path   f_path;
```

```
4     struct inode     *f_inode; /* cached value */
5     const struct file_operations  *f_op;
6     ...
7     atomic_long_t    f_count;
8     unsigned int     f_flags;
9     fmode_t      f_mode;
10    struct mutex     f_pos_lock;
11    loff_t       f_pos;
12    ...
13    struct address_space  *f_mapping;
14    ...
15  } __randomize_layout
16    __attribute__((aligned(4)));
```

2.7.5 union i2c_smbus_data

union i2c_smbus_data — сообщение, передаваемое по шине I2C.

Листинг 2.5 – union i2c_smbus_data

```
1  #define I2C_SMBUS_BLOCK_MAX 32  /* As specified in SMBus
       standard */
2  union i2c_smbus_data {
3    __u8 byte;
4    __u16 word;
5    __u8 block[I2C_SMBUS_BLOCK_MAX + 2]; /* block[0] is used for
         length */
6                 /* and one more for user-space compatibility */
7  };
```

2.7.6 struct i2c_smbus_ioctl_data

struct i2c_smbus_ioctl_data — структура, использующаяся в вызове ioctl.

Листинг 2.6 – struct i2c_smbus_ioctl_data

```
1  /* This is the structure as used in the I2C_SMBUS ioctl call */
2  struct i2c_smbus_ioctl_data {
3    __u8 read_write;
4    __u8 command;
5    __u32 size;
6    union i2c_smbus_data __user *data;
7  };
```

2.7.7 struct i2c_client

struct i2c_client представляет ведомое устройство I2C, подключенное к шине. addr — адрес на I2C шине, name — имя чипа устройства, идентифицирует тип устройства, dev — структура-описатель устройства, adapter — сегмент шины, к которому подключено устройство.

Листинг 2.7 – struct i2c_client

```
1  struct i2c_client {
2    unsigned short flags;    /* div., see below     */
3    unsigned short addr;     /* chip address - NOTE: 7bit  */
4            /* addresses are stored in the  */
5            /* _LOWER_ 7 bits    */
6    char name[I2C_NAME_SIZE];
7    struct i2c_adapter *adapter;  /* the adapter we sit on  */
8    struct device dev;     /* the device structure    */
9    int init_irq;      /* irq set at initialization  */
10   int irq;       /* irq issued by device    */
11   struct list_head detected;
12 #if IS_ENABLED(CONFIG_I2C_SLAVE)
13   i2c_slave_cb_t slave_cb;  /* callback for slave mode  */
14 #endif
15   void *devres_group_id;     /* ID of probe devres group */
16 };
```

2.8 Точки входа драйвера

Драйвер имеет следующие точки входа:

1) `my_init` — инициализация модуля;

2) `my_exit` — завершение работы модуля;

3) `dev_read` — чтение из специального файла устройства;

4) `dev_write` — запись в специальный файл устройства.

2.9 Взаимодействие модулей ПО

На рисунке 2.8 представлена схема взаимодействия модулей ПО.

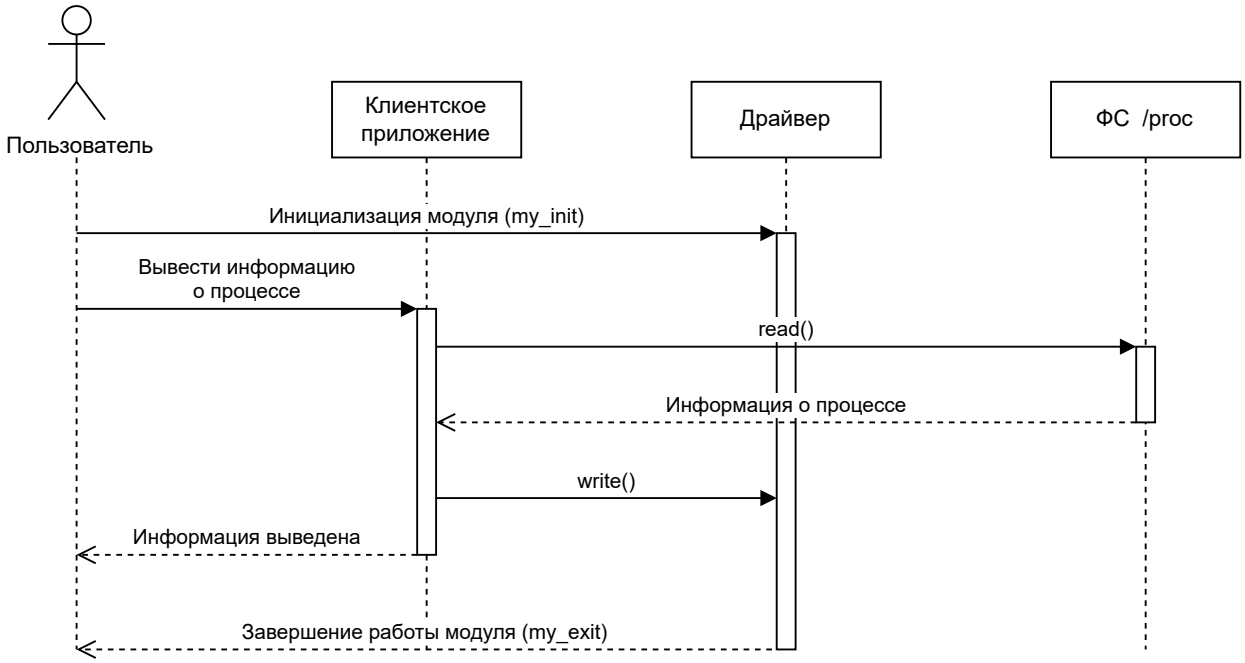

Рисунок 2.8 – Схема взаимодействия модулей ПО

Вывод

Были составлены схемы алгоритмов инициализации модуля ядра — драйвера символьного дисплея, вывода данных на символьный дисплей, чтения данных из устройства, записи данных в шину через интерфейс I2C и работы клиентского приложения. Были приведены необходимые для разработки структуры ядра, описаны точки входа драйвера символьного дисплея. Также, была представлена схема взаимодействия модулей программного комплекса.

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для реализации загружаемого модуля был выбран язык Си, так как необходимо написать загружаемый модуль ядра, и ядро Linux реализовано на языке C. Операционная система Linux позволяет писать загружаемые модули ядра на Rust и на Си, но Rust обладает определёнными ограничениями в написании модулей ядра.

В качестве среды разработки выбрана CLion: запуск программного комплекса осуществлялся на компьютере, к которому было выполнено подключение по ssh, и данная среда поддерживает передачу данных по ssh из графического интерфейса.

3.2 Реализация загружаемого модуля ядра — драйвера символьного дисплея

В листинге 3.1 представлены реализации функции инициализации и завершения работы загружаемого модуля ядра — драйвера символьного дисплея.

Листинг 3.1 – Реализация загружаемого модуля ядра — драйвера символьного дисплея

```
1  static const struct file_operations my_fops = {
2          .owner   = THIS_MODULE,
3          .read    = dev_read,
4          .write   = dev_write,
5  };
6
7  static struct miscdevice my_dev = {
8          MISC_DYNAMIC_MINOR,
9          "proclcd",
10         &my_fops
11 };
12
13 static int __init my_init(void)
14 {
15     int ret;
16
17     rc = misc_register(&my_dev);
```

```
18      if (rc) {
19          printk(KERN_ERR "+ unable to register misc device\n");
20      }
21
22      start();
23
24      printk(KERN_DEBUG "+ module loaded");
25
26      return ret;
27  }
28
29  static void __exit my_exit(void)
30  {
31      misc_deregister( &my_dev );
32      printk(KERN_DEBUG "+ module unloaded");
33  }
34
35  module_init(my_init);
36  module_exit(my_exit);
```

В листинге 3.2 представлена реализация функции установления связи с шиной I2C и настройки соединения с дисплеем.

Листинг 3.2 – Реализация функции установления связи с шиной I2C и настройки соединения с дисплеем

```
1   int start(void) {
2       int length;
3       unsigned char buffer[60] = {0};
4
5       printk(KERN_DEBUG "+ opening");
6
7       char *filename = (char*)"/dev/i2c-5";
8       file_i2c = filp_open(filename, O_RDWR, 0);
9       if (!file_i2c) {
10          printk(KERN_DEBUG "+ failed to open the i2c bus");
11          return 1;
12      }
13
14      printk(KERN_DEBUG "+ configuring");
15
16      if ((configure()) != 0) {
17          printk(KERN_DEBUG "+ failed to configure");
```

```
18          return 1;
19      }
20
21      printk(KERN_DEBUG "+ setting cursor");
22
23      set_cursor(0, 0);
24
25      delay_microseconds(100000);
26
27      return 0;
28  }
```

В листинге 3.3 представлена реализация функции вывода данных на дисплей.

Листинг 3.3 – Реализация функции вывода данных на дисплей

```
1  static ssize_t dev_write(struct file *file, const char *buf,
       size_t count, loff_t *ppos) {
2      char data[1024];
3      if (copy_from_user(data, buf, strlen(buf))) {
4          return -EINVAL;
5      }
6
7      str_pos = 0;
8      col_pos = 0;
9
10     for (int i = 0; i < count; i++) {
11         if ((col_pos==0) && (str_pos==0)) {
12             clear();
13         }
14
15         set_cursor(col_pos, str_pos);
16         delay_microseconds(10000);
17
18         printk(KERN_DEBUG "+ cursor col %d row %d cur sym %c",
               col_pos, str_pos, data[i]);
19
20         if (buf[i] != '\n') {
21             print_byte(data[i]);
22             col_pos++;
23             delay_microseconds(10000);
24         } else {
```

```
25              col_pos=16;
26          }
27
28          if (col_pos == 16) {
29              col_pos = 0;
30              str_pos++;
31              if (str_pos == 2) {
32                  str_pos = 0;
33              }
34          }
35      }
36
37      return count;
38  }
```

В листинге 3.4 представлена реализация функции передачи байта данных с помощью интерфейса I2C.

Листинг 3.4 – Реализация функции передачи байта данных с помощью интерфейса I2C

```
1  int write_byte_data(int reg, int data, const int addr) {
2      if (vfs_ioctl(file_i2c, I2C_SLAVE, addr) < 0)
3      {
4          printk(KERN_DEBUG "+ failed to acquire bus access
               and/or talk to slave");
5          return 1;
6      }
7
8      union i2c_smbus_data msg;
9      msg.byte = data;
10
11     struct i2c_smbus_ioctl_data req = {
12         .read_write = 0,
13         .command = reg,
14         .size = 2,
15         .data = &msg,
16     };
17
18     struct i2c_client *client = file_i2c->private_data;
19     int res = i2c_smbus_xfer(client->adapter, client->addr,
           client->flags,
20                              req.read_write, req.command, req.size,
```

```
                                &msg);
    if (res < 0) {
        printk(KERN_DEBUG "+ failed to smbus_xfer %d", res);
        return 1;
    }

    return 0;
}
```

В листинге 3.5 представлен Makefile загружаемого модуля ядра.

Листинг 3.5 – Makefile загружаемого модуля ядра

```
ifneq ($(KERNELRELEASE),)
  obj-m := my_driver.o
else
  CURRENT = $(shell uname -r)
  KDIR = /lib/modules/$(CURRENT)/build
  PWD = $(shell pwd)

default:
  $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
  rm -rf .tmp_versions
  rm *.ko
  rm *.o
  rm *.mod.c
  rm *.symvers
  rm *.order

endif
```

3.3 Реализация клиентского приложения

В листинге 3.6 представлена реализация функции-обработчика запроса пользователя для вывода из proc данных о потребляемой процессом виртуальной памяти.

Листинг 3.6 – Реализация функции-обработчика запроса пользователя для вывода из proc данных о потребляемой процессом виртуальной памяти

```
void vm_handler() {
    char path[PATH_MAX];
```

```
3       snprintf(path, PATH_MAX, "/proc/%d/statm", pid);
4
5       int dev = open(DEV_FILE, O_RDWR);
6
7       FILE *statm = fopen(path, "r");
8
9       char buf[BUF_SIZE + 1] = "\0";
10      int len, n;
11
12      while (stop == 0) {
13          fscanf(statm, "%d", &n);
14
15          char str[1024];
16          sprintf(str, "%d MB", n * sysconf(_SC_PAGE_SIZE) / 1024
                / 1024);
17          write(dev, str, strlen(str));
18
19          fseek(statm, 0, SEEK_SET);
20          sleep(3);
21      }
22
23      stop = 0;
24
25      fclose(statm);
26      close(dev);
27  }
```

Вывод

Был обоснован выбор языка и среды программирования и приведены листинги кода. Представлены листинги функций инициализации и завершения работы драйвера дисплея, функции установления связи с шиной I2C и настройки соединения с дисплеем. Также приведены листинги общей функции вывода данных на дисплей и передачи байта данных с помощью интерфейса I2C.

4 Исследовательский раздел

Характеристики компьютера, на котором было проведено тестирование разработанного ПО: операционная система Debian Bookworm, версия ядра Linux 6.1.31.

На рисунке 4.1 представлен пример отображения на дисплее информации о потребляемой процессом containerd виртуальной памяти.

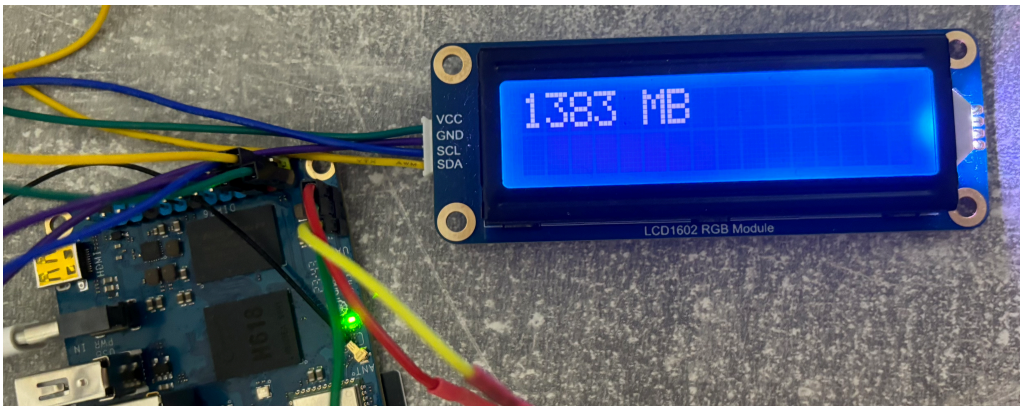

Рисунок 4.1 – Пример отображения на дисплее информации о потребляемой процессом containerd виртуальной памяти

На рисунках 4.2 – 4.3 представлен пример отображения в системе специального файла устройства для разработанного драйвера и модуля ядра соответственно.

```
orangepi@orangepizero3:~/code$ ls -la /dev | grep proclcd
crw-rw-rw-   1 root     root     10, 121 Jan 21 22:46 proclcd
```

Рисунок 4.2 – Пример отображения в системе специального файла устройства для разработанного драйвера

```
orangepi@orangepizero3:~/code$ sudo lsmod | grep my_driver
my_driver              16384  0
```

Рисунок 4.3 – Пример отображения в системе модуля ядра

На рисунках 4.4 – 4.5 представлен пример логов разработанного модуля ядра при инициализации и в процессе вывода данных на дисплей соответственно.

```
[30603.293853] + module unloaded
[30622.143528] + opening
[30622.143601] + configuring
[30622.143611] + set show_function 1st try
[30622.249094] + set show_function 2nd try
[30622.357067] + set show_function 3rd try
[30622.357389] + set show_function 4th try
[30622.465065] + turn the display on with no cursor or blinking default
[30622.465384] + write to send command clear
[30622.573087] + set text direction and entry mode
[30622.573428] + setting colors
[30622.575333] + ending configuration
[30622.681085] + module loaded
```

Рисунок 4.4 – Пример логов разработанного модуля ядра при инициализации

```
[45024.824360] + cursor col 5 row 0 cur sym M
[45024.864385] + cursor col 6 row 0 cur sym B
[45027.904355] + cursor col 0 row 0 cur sym 1
[45027.944354] + cursor col 1 row 0 cur sym 3
[45027.984353] + cursor col 2 row 0 cur sym 8
[45028.024353] + cursor col 3 row 0 cur sym 3
[45028.064353] + cursor col 4 row 0 cur sym
[45028.104353] + cursor col 5 row 0 cur sym M
[45028.144357] + cursor col 6 row 0 cur sym B
[45031.184370] + cursor col 0 row 0 cur sym 1
[45031.224346] + cursor col 1 row 0 cur sym 3
[45031.264346] + cursor col 2 row 0 cur sym 8
[45031.304346] + cursor col 3 row 0 cur sym 3
[45031.344346] + cursor col 4 row 0 cur sym
[45031.384341] + cursor col 5 row 0 cur sym M
```

Рисунок 4.5 – Пример логов разработанного модуля ядра в процессе вывода данных на дисплей

На рисунке 4.6 представлен интерфейс клиентской части приложения.

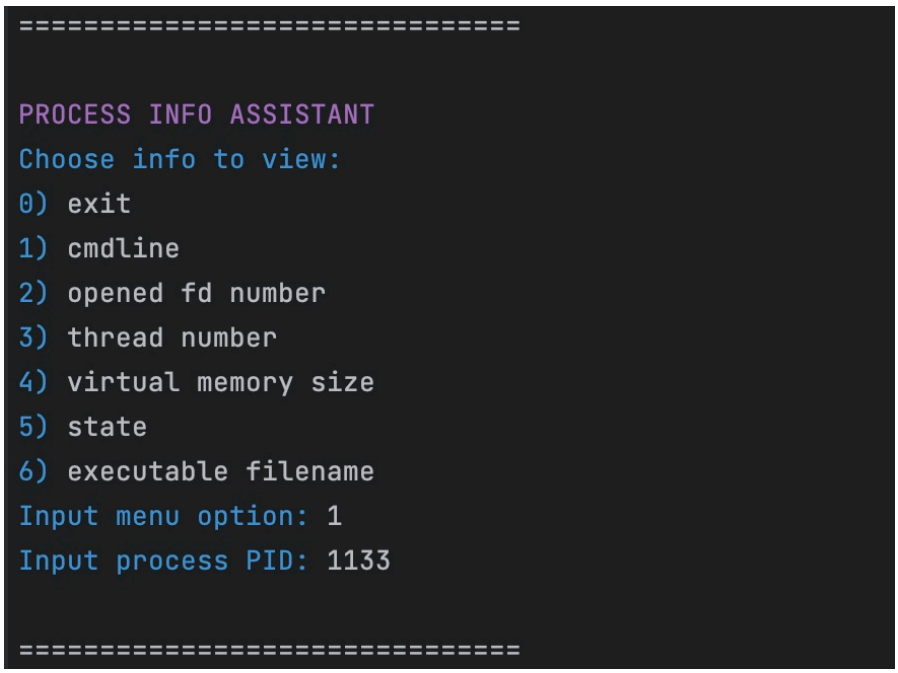

Рисунок 4.6 – Интерфейс клиентской части приложения

Вывод

Был представлен пример отображения информации о конкретном процессе на символьном дисплее. Также, были приведены примеры отображения в системе специального файла устройства, отображение модуля ядра, пример отладочного вывода разработанного модуля и вид пользовательского интерфейса.

ЗАКЛЮЧЕНИЕ

В ходе работы были решены следующие задачи:

1) Проведён сравнительный анализ способов реализации драйверов устройств.

2) Сделан выбор способа реализации драйвера символьного дисплея.

3) Проведён сравнительный анализ существующих интерфейсов взаимодействия с символьным жидкокристаллическим дисплеем.

4) Сделан выбор интерфейса взаимодействия с символьным жидкокристаллическим дисплеем в соответствии с поставленной целью.

5) Сделан выбор системных вызовов, поддерживаемых драйвером, в соответствии с поставленной целью.

6) Разработаны алгоритмы и структура ПО.

7) Реализован драйвер символьного дисплея для вывода информации о конкретном процессе как загружаемый модуль ядра Linux.

8) Реализована программа уровня пользователя для взаимодействия с символьным дисплеем.

При выполнении курсовой работы было принято решение реализовать драйвер символьного дисплея для вывода информации о процессе как модуль ядра Linux и использовать вспомогательный интерфейс I2C. Для получения данных о процессе по PID использована виртуальная файловая система proc.

Таким образом, была достигнута цель работы: реализован программный комплекс, состоящий из загружаемого модуля ядра Linux и программы уровня пользователя для вывода на символьный дисплей заданной приложением информации о конкретном процессе. В ходе проведения исследования продемонстрирована корректность работы разработанного ПО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Jonathan Corbet A. R.*, *Kroah-Hartman G.* Linux Device Drivers. — 2005.
2. *ХЕЙС П. Д.*, *ЭР К. Е.* РЕГУЛИРОВКА СИНХРОНИЗАЦИИ СТЕКА ДЛЯ ПОСЛЕДОВАТЕЛЬНОЙ СВЯЗИ. — 2019.
3. *Занина В.*, *Иванова Е.* Программная реализация интерфейса UART // Современные научные исследования и разработки. — 2018. — Т. 2, № 11. — С. 269—270.
4. Программный драйвер для работы с разными видами интерфейсов SPI / Э. Сёмка [и др.] // Вопросы радиоэлектроники. — 2020. — Т. 49, № 9. — С. 38—45.
5. *Ванройе Н.*, *Ечеистов В.* Особенности передачи данных по шине I2C // Аллея науки. — 2018. — Т. 4, № 6. — С. 289—293.
6. Miscellaneous Character Drivers [Электронный ресурс]. — Режим доступа: `https://www.linuxjournal.com/article/2920` (дата обращения: 13.01.2024).

ПРИЛОЖЕНИЕ А
Исходный код

В листинге А.1 приведен код разработанного модуля ядра — драйвера символьного дисплея.

Листинг А.1 – Код модуля ядра — драйвера символьного дисплея

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <asm/io.h>
#include <linux/unistd.h>
#include <linux/delay.h>
#include <asm/uaccess.h>
#include <linux/miscdevice.h>
#include <linux/fcntl.h>
#include <linux/ioctl.h>
#include <linux/i2c-dev.h>
#include <linux/types.h>
#include <linux/delay.h>
#include <linux/i2c.h>
#include <uapi/linux/errno.h>
#include <asm/errno.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Karpova Ekaterina");

#define LCD_ADDRESS     (0x3e)
#define RGB_ADDRESS     (0xc0>>1)
#define COMMAND_REG 0x80
#define DATA_REG 0x40

#define REG_RED     0x04
#define REG_GREEN       0x03
#define REG_BLUE        0x02
#define REG_MODE1       0x00
#define REG_MODE2       0x01
#define REG_OUTPUT      0x08
#define LCD_CLEARDISPLAY  0x01
#define LCD_ENTRYMODESET  0x04
#define LCD_DISPLAYCONTROL  0x08
#define LCD_FUNCTIONSET  0x20
```

```
36
37 //#flags for display entry mode
38 #define LCD_ENTRYLEFT  0x02
39 #define LCD_ENTRYSHIFTDECREMENT  0x00
40
41 //#flags for display on/off control
42 #define LCD_DISPLAYON  0x04
43 #define LCD_CURSOROFF  0x00
44
45 //#flags for function set
46 #define LCD_4BITMODE  0x00
47 #define LCD_2LINE  0x08
48
49 struct file* file_i2c;
50
51 static inline int delay_microseconds(int value) {
52     if (value > 1000) {
53         msleep(value/1000);
54     }
55     udelay(value % 1000);
56     return 0;
57 }
58
59 int write_byte_data(int reg, int data, const int addr) {
60     if (vfs_ioctl(file_i2c, I2C_SLAVE, addr) < 0)
61     {
62         printk(KERN_DEBUG "+ failed to acquire bus access
              and/or talk to slave");
63         return 1;
64     }
65
66     union i2c_smbus_data msg;
67     msg.byte = data;
68
69     struct i2c_smbus_ioctl_data req = {
70         .read_write = 0,
71         .command = reg,
72         .size = 2,
73         .data = &msg,
74     };
75
```

```
76      struct i2c_client *client = file_i2c->private_data;
77      int res = i2c_smbus_xfer(client->adapter, client->addr,
            client->flags,
78                               req.read_write, req.command, req.size,
                                   &msg);
79      if (res < 0) {
80          printk(KERN_DEBUG "+ failed to smbus_xfer %d", res);
81          return 1;
82      }
83
84      return 0;
85  }
86
87  int command(int cmd) {
88      if ((write_byte_data(COMMAND_REG, cmd, LCD_ADDRESS)) != 0)
89      {
90          printk(KERN_DEBUG "+ failed to write byte data");
91          return 1;
92      }
93
94      return 0;
95  }
96
97  int print_byte(int data) {
98      if ((write_byte_data(DATA_REG, data, LCD_ADDRESS)) != 0)
99      {
100         printk(KERN_DEBUG "+ failed to write byte data");
101         return 1;
102     }
103
104     return 0;
105 }
106
107 int set_reg(int reg, int data) {
108     if ((write_byte_data(reg, data, RGB_ADDRESS)) != 0)
109     {
110         printk(KERN_DEBUG "+ failed to write byte data");
111         return 1;
112     }
113
114     return 0;
```

```
115 }
116
117 void set_cursor(int col, int row) {
118     if (row == 0) {
119         col |= 0x80;
120     } else {
121         col |= 0xc0;
122     }
123
124     if ((command(col)) != 0) {
125         printk(KERN_DEBUG "+ failed to set cursor");
126     }
127 }
128
129 int configure(void) {
130     printk(KERN_DEBUG "+ set show_function 1st try");
131
132     if ((command(LCD_4BITMODE | LCD_2LINE | LCD_FUNCTIONSET))
        != 0) {
133         printk(KERN_DEBUG "+ failed to set show_function 1st
            try");
134         return 1;
135     }
136
137     delay_microseconds(100000);
138
139     printk(KERN_DEBUG "+ set show_function 2nd try");
140
141     if ((command(LCD_4BITMODE | LCD_2LINE | LCD_FUNCTIONSET))
        != 0) {
142         printk(KERN_DEBUG "+ failed to set show_function 2nd
            try");
143         return 1;
144     }
145
146     delay_microseconds(100000);
147
148     printk(KERN_DEBUG "+ set show_function 3rd try");
149
150     if ((command(LCD_4BITMODE | LCD_2LINE | LCD_FUNCTIONSET))
        != 0) {
```

```
151         printk(KERN_DEBUG "+ failed to set show_function 3rd
                try");
152         return 1;
153     }
154
155     printk(KERN_DEBUG "+ set show_function 4th try");
156
157     if ((command(LCD_4BITMODE | LCD_2LINE | LCD_FUNCTIONSET))
           != 0) {
158         printk(KERN_DEBUG "+ failed to set show_function 4th
                try");
159         return 1;
160     }
161
162     delay_microseconds(100000);
163
164     printk(KERN_DEBUG "+ turn the display on with no cursor or
           blinking default");
165
166     if ((command(LCD_DISPLAYON | LCD_CURSOROFF |
           LCD_DISPLAYCONTROL)) != 0) {
167         printk(KERN_DEBUG "+ failed to turn the display on with
                no cursor or blinking default");
168         return 1;
169     }
170
171     printk("+ write to send command clear");
172
173     if ((command(LCD_CLEARDISPLAY)) != 0) {
174         printk(KERN_DEBUG "+ failed to write to send command
                clear");
175         return 1;
176     }
177
178     delay_microseconds(100000);
179
180     printk(KERN_DEBUG "+ set text direction and entry mode");
181
182     if ((command(LCD_ENTRYLEFT | LCD_ENTRYSHIFTDECREMENT |
           LCD_ENTRYMODESET)) != 0) {
183         printk(KERN_DEBUG "+ failed to set text direction and
```

```
            entry mode");
184         return 1;
185     }
186
187     printk(KERN_DEBUG "+ setting colors");
188
189     set_reg(REG_MODE1, 0);
190     set_reg(REG_OUTPUT, 0xFF);
191     set_reg(REG_MODE2, 0x20);
192
193     set_reg(REG_RED, 255);
194     set_reg(REG_GREEN, 255);
195     set_reg(REG_BLUE, 255);
196
197     printk(KERN_DEBUG "+ ending configuration");
198
199     return 0;
200 }
201
202 int clear(void) {
203     if ((command(LCD_CLEARDISPLAY)) != 0) {
204         printk(KERN_DEBUG "+ failed to write to send command");
205         return 1;
206     }
207
208     return 0;
209 }
210
211 int print_string(char* str) {
212     for (int i = 0; str[i] != 0; i++) {
213         if (i > 0 && i % 16 == 0) {
214             set_cursor(0, 1);
215         }
216         if ((print_byte(str[i])) != 0) {
217             printk(KERN_DEBUG "+ Failed to write data");
218             return 1;
219         }
220     }
221
222     return 0;
223 }
```

```
224
225 int start(void) {
226     int length;
227     unsigned char buffer[60] = {0};
228
229     printk(KERN_DEBUG "+ opening");
230
231     char *filename = (char*)"/dev/i2c-5";
232     file_i2c = filp_open(filename, O_RDWR, 0);
233     if (!file_i2c) {
234         printk(KERN_DEBUG "+ failed to open the i2c bus");
235         return 1;
236     }
237
238     printk(KERN_DEBUG "+ configuring");
239
240     if ((configure()) != 0) {
241         printk(KERN_DEBUG "+ failed to configure");
242         return 1;
243     }
244
245     printk(KERN_DEBUG "+ setting cursor");
246
247     set_cursor(0, 0);
248
249     delay_microseconds(100000);
250
251     return 0;
252 }
253
254 static char *info_str = "LCD display driver.";
255
256 static ssize_t dev_read( struct file * file, char * buf, size_t
       count, loff_t *ppos) {
257     int len = strlen(info_str);
258
259     if(count < len) {
260         return -EINVAL;
261     }
262
263     if (*ppos != 0) {
```

```
264         return 0;
265     }
266
267     if (copy_to_user(buf, info_str, len)) {
268         return -EINVAL;
269     }
270
271     *ppos = len;
272     return len;
273 }
274
275 static int str_pos = 0;
276 static int col_pos = 0;
277
278 static ssize_t dev_write(struct file *file, const char *buf,
       size_t count, loff_t *ppos) {
279     str_pos = 0;
280     col_pos = 0;
281
282     for (size_t i = 0; i < count; i++) {
283         if ((col_pos==0) && (str_pos==0)) {
284             clear();
285         }
286
287         set_cursor(col_pos, str_pos);
288         delay_microseconds(10000);
289
290         printk(KERN_DEBUG "+ cursor col %d row %d cur sym %c",
              col_pos, str_pos, buf[i]);
291
292         if (buf[i] != '\n') {
293             print_byte(buf[i]);
294             col_pos++;
295             delay_microseconds(10000);
296         } else {
297             col_pos=16;
298         }
299
300         if (col_pos == 16) {
301             col_pos = 0;
302             str_pos++;
```

```
303             if (str_pos == 2) {
304                 str_pos = 0;
305             }
306         }
307     }
308
309     return count;
310 }
311
312 static const struct file_operations my_fops = {
313         .owner  = THIS_MODULE,
314         .read   = dev_read,
315         .write  = dev_write,
316 };
317
318 static struct miscdevice my_dev = {
319         MISC_DYNAMIC_MINOR,
320         "proclcd",
321         &my_fops
322 };
323
324 static int __init my_init(void)
325 {
326     int ret;
327
328     rc = misc_register(&my_dev);
329     if (rc) {
330         printk(KERN_ERR "+ unable to register misc device\n");
331     }
332
333     start();
334
335     printk(KERN_DEBUG "+ module loaded");
336
337     return ret;
338 }
339
340 static void __exit my_exit(void)
341 {
342     misc_deregister( &my_dev );
343     printk(KERN_DEBUG "+ module unloaded");
```

```
344 }
345
346 module_init(my_init);
347 module_exit(my_exit);
```

В листинге А.2 приведен код разработанного приложения уровня пользователя.

Листинг А.2 – Код разработанного приложения уровня пользователя

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <dirent.h>
5  #include <errno.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <stdint.h>
9  #include <signal.h>
10 #include "client.h"
11 #include <sys/stat.h>
12
13 #define DEV_FILE "/dev/proclcd"
14 #define _POSIX1_SOURCE 2
15 #define BUF_SIZE 102400
16 #define ANSI_COLOR_RED     "\x1b[31m"
17 #define ANSI_COLOR_GREEN   "\x1b[32m"
18 #define ANSI_COLOR_YELLOW  "\x1b[33m"
19 #define ANSI_COLOR_BLUE    "\x1b[34m"
20 #define ANSI_COLOR_MAGENTA "\x1b[35m"
21 #define ANSI_COLOR_CYAN    "\x1b[36m"
22 #define ANSI_COLOR_RESET   "\x1b[0m"
23
24 int pid = 1;
25 int opt = 0;
26 int stop = 0;
27
28 int existDir(const char * name)
29 {
30     struct stat s;
31     if (stat(name,&s)) return 0;
32     return S_ISDIR(s.st_mode);
33 };
```

```
34
35 int print_menu() {
36     int rc = EXIT_SUCCESS;
37
38     printf("\n\n==========================================\n");
39     printf("\n" ANSI_COLOR_MAGENTA "PROCESS INFO ASSISTANT"
          ANSI_COLOR_RESET "\n");
40
41     printf(ANSI_COLOR_BLUE "Choose info to view:"
          ANSI_COLOR_RESET "\n");
42     printf(ANSI_COLOR_BLUE "0) " ANSI_COLOR_RESET "exit\n");
43     printf(ANSI_COLOR_BLUE "1) " ANSI_COLOR_RESET "cmdline\n");
44     printf(ANSI_COLOR_BLUE "2) " ANSI_COLOR_RESET "opened fd
          number\n");
45     printf(ANSI_COLOR_BLUE "3) " ANSI_COLOR_RESET "thread
          number\n");
46     printf(ANSI_COLOR_BLUE "4) " ANSI_COLOR_RESET "virtual
          memory size\n");
47     printf(ANSI_COLOR_BLUE "5) " ANSI_COLOR_RESET "state\n");
48     printf(ANSI_COLOR_BLUE "6) " ANSI_COLOR_RESET "executable
          filename\n");
49
50     printf(ANSI_COLOR_BLUE "Input menu option: "
          ANSI_COLOR_RESET);
51     if ((rc = scanf("%d", &opt)) != 1 || opt < 0 || opt > 6) {
52         printf(ANSI_COLOR_RED "invalid menu option"
              ANSI_COLOR_RESET "\n");
53         printf("\n==========================================\n");
54         return EXIT_FAILURE;
55     }
56
57     if (opt != 0) {
58         printf(ANSI_COLOR_BLUE "Input process PID: "
              ANSI_COLOR_RESET);
59         if ((rc = scanf("%d", &pid)) != 1) {
60             printf(ANSI_COLOR_RED "invalid process pid"
                  ANSI_COLOR_RESET "\n");
61             printf("\n==========================================\n");
62             return EXIT_FAILURE;
63         }
64
```

```
65          char str[1024];
66          sprintf(str, "/proc/%d", pid);
67
68          if (existDir(str) == 0) {
69              printf(ANSI_COLOR_RED "process doesn't exist"
                    ANSI_COLOR_RESET "\n");
70              printf("\n=================================\n");
71              return EXIT_FAILURE;
72          }
73      }
74
75      printf("\n=================================\n");
76      return EXIT_SUCCESS;
77  }
78
79  void exit_handler() {
80      opt = -1;
81      printf("\n" ANSI_COLOR_YELLOW "program exiting"
           ANSI_COLOR_RESET "\n");
82  }
83
84  void cmdline_handler() {
85      char buf[BUF_SIZE+1] = "\0";
86      int len, i;
87      FILE* f;
88
89      int dev = open(DEV_FILE, O_RDWR);
90
91      char path[PATH_MAX];
92      snprintf(path, PATH_MAX, "/proc/%d/cmdline", pid);
93
94      f = fopen(path, "r");
95
96      while (stop == 0) {
97          while ((len = fread(buf, 1, BUF_SIZE, f)) > 0)
98          {
99              buf[len] = '\0';
100             write(dev, buf, len-1);
101         }
102
103         fseek(f, 0, SEEK_SET);
```

```
104             sleep(3);
105         }
106
107         stop = 0;
108
109         fclose(f);
110         close(dev);
111     }
112
113     void fds_handler() {
114         char path[PATH_MAX];
115         snprintf(path, PATH_MAX, "/proc/%d/fd", pid);
116
117         int dev = open(DEV_FILE, O_RDWR);
118
119         struct dirent *d;
120         DIR *dh = opendir(path);
121         if (!dh)
122         {
123             perror("open task dir\n");
124             exit(1);
125         }
126
127         long pos = telldir(dh);
128
129         while (stop == 0) {
130             int fd_num = 0;
131             while ((d = readdir(dh)) != NULL)
132             {
133                 if (d->d_name[0] == '.')
134                     continue;
135
136                 fd_num++;
137             }
138
139             char str[1024];
140             sprintf(str, "%d", fd_num);
141             write(dev, str, strlen(str));
142
143             seekdir(dh,pos);
144
```

```
145         sleep(3);
146     }
147
148     stop = 0;
149
150     closedir(dh);
151     close(dev);
152 }
153
154 void threads_handler() {
155     char path[PATH_MAX];
156     snprintf(path, PATH_MAX, "/proc/%d/task", pid);
157
158     int dev = open(DEV_FILE, O_RDWR);
159
160     struct dirent *d;
161     DIR *dh = opendir(path);
162     if (!dh)
163     {
164         perror("open task dir\n");
165         exit(1);
166     }
167
168     long pos = telldir(dh);
169
170     while (stop == 0) {
171         int thread_num = 0;
172         while ((d = readdir(dh)) != NULL)
173         {
174             if (d->d_name[0] == '.')
175                 continue;
176
177             thread_num++;
178         }
179
180         char str[1024];
181         sprintf(str, "%d", thread_num);
182         write(dev, str, strlen(str));
183
184         seekdir(dh,pos);
185
```

```
186             sleep(3);
187         }
188
189         stop = 0;
190
191         closedir(dh);
192         close(dev);
193     }
194
195     void vm_handler() {
196         char path[PATH_MAX];
197         snprintf(path, PATH_MAX, "/proc/%d/statm", pid);
198
199         int dev = open(DEV_FILE, O_RDWR);
200
201         FILE *statm = fopen(path, "r");
202
203         char buf[BUF_SIZE + 1] = "\0";
204         int len, n;
205
206         while (stop == 0) {
207             fscanf(statm, "%d", &n);
208
209             char str[1024];
210             sprintf(str, "%d MB", n * sysconf(_SC_PAGE_SIZE) / 1024
                    / 1024);
211             write(dev, str, strlen(str));
212
213             fseek(statm, 0, SEEK_SET);
214             sleep(3);
215         }
216
217         stop = 0;
218
219         fclose(statm);
220         close(dev);
221     }
222
223     void state_handler() {
224         char path[PATH_MAX];
225         snprintf(path, PATH_MAX, "/proc/%d/stat", pid);
```

```
226
227     int dev = open(DEV_FILE, O_RDWR);
228
229     FILE *stat = fopen(path, "r");
230
231     char buf[BUF_SIZE + 1] = "\0";
232     int len, n;
233     char name[1024];
234     char state;
235
236     while (stop == 0) {
237         fscanf(stat, "%d %s %c", &n, name, &state);
238
239         char str[1024];
240         if (state == 'R') {
241             sprintf(str, "%c - runnable", state);
242         } else if (state == 'D') {
243             sprintf(str, "%c - uninterruptable", state);
244         } else if (state == 'T') {
245             sprintf(str, "%c - stopped", state);
246         } else if (state == 'S') {
247             sprintf(str, "%c - sleeping", state);
248         } else if (state == 'Z') {
249             sprintf(str, "%c - zombie", state);
250         } else if (state == '<') {
251             sprintf(str, "%c - negative nice", state);
252         } else if (state == 'N') {
253             sprintf(str, "%c - positive nice", state);
254         }
255
256         write(dev, str, strlen(str));
257
258         fseek(stat, 0, SEEK_SET);
259         sleep(3);
260     }
261
262     stop = 0;
263
264     fclose(stat);
265     close(dev);
266 }
```

```
267
268 void comm_handler() {
269     char buf[BUF_SIZE+1] = "\0";
270     int len, i;
271     FILE* f;
272
273     int dev = open(DEV_FILE, O_RDWR);
274
275     char path[PATH_MAX];
276     snprintf(path, PATH_MAX, "/proc/%d/comm", pid);
277
278     f = fopen(path, "r");
279
280     while (stop == 0) {
281         while ((len = fread(buf, 1, BUF_SIZE, f)) > 0)
282         {
283             buf[len] = '\0';
284             write(dev, buf, len-1);
285         }
286
287         fseek(f, 0, SEEK_SET);
288         sleep(3);
289     }
290
291     stop = 0;
292
293     fclose(f);
294     close(dev);
295 }
296
297 void signal_handler(int signal)
298 {
299     printf("\n" ANSI_COLOR_YELLOW "caught signal = %d"
          ANSI_COLOR_RESET "\n", signal);
300     stop = 1;
301 }
302
303 int main(int argc, char **argv) {
304     if ((signal(SIGINT, signal_handler) == SIG_ERR)) {
305         perror("Can't attach handler\n");
306         return EXIT_FAILURE;
```

```
307     }
308
309     if (argc != 1) {
310         printf(ANSI_COLOR_RED "no arguments needed"
              ANSI_COLOR_RESET "\n");
311         return EXIT_FAILURE;
312     }
313
314     void (*handlers[7])(void);
315     handlers[0] = &exit_handler;
316     handlers[1] = &cmdline_handler;
317     handlers[2] = &fds_handler;
318     handlers[3] = &threads_handler;
319     handlers[4] = &vm_handler;
320     handlers[5] = &state_handler;
321     handlers[6] = &comm_handler;
322
323     int rc;
324     while (opt >= 0) {
325         rc = print_menu();
326         if (rc != EXIT_SUCCESS) {
327             printf(ANSI_COLOR_RED "menu error" ANSI_COLOR_RESET
                  "\n");
328             opt = 0;
329             pid = 1;
330             stop = 0;
331         } else {
332             handlers[opt]();
333         }
334     }
335
336     return EXIT_SUCCESS;
337 }
```