



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 (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## *К КУРСОВОЙ РАБОТЕ*

### *НА ТЕМУ:*

«Мониторинг информации об операционной системе  
Linux и ее процессах»

Студент ИУ7-73Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

И. А. Цветков  
(И. О. Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

Н. Ю. Рязанова  
(И. О. Фамилия)

*2023 г.*

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ</b>	<b>5</b>
<b>1 Аналитическая часть</b>	<b>6</b>
1.1 Постановка задачи	6
1.2 Способы получения информации о системе	6
1.3 Получение информации о системе	7
1.3.1 Имя компьютера	8
1.3.2 Информация о процессоре	8
1.3.3 Информация о видеокарте	8
1.3.4 Версия ядра операционной системы	8
1.3.5 Загруженность процессора	8
1.3.6 Информация об оперативной памяти компьютера	9
1.4 Загружаемый модуль ядра	9
1.4.1 Загрузка и выгрузка загружаемого модуля ядра	10
1.5 Взаимодействие пространства ядра и пользователя	10
1.6 Получение информации о процессах	11
1.6.1 Предоставление процессов в памяти	12
1.6.2 Построение дерева процессов	12
1.6.3 Получение информации о памяти процесса	12
1.6.4 Получение информации об открытых процессом файлах	13
<b>2 Конструкторская часть</b>	<b>14</b>
2.1 Диграмма состояний модуля ядра	14
2.2 Структура программного обеспечения	15
2.3 Схемы алгоритмов функций загружаемого модуля ядра	15
2.4 Пользовательское приложение	15
<b>3 Технологическая часть</b>	<b>18</b>
3.1 Средства реализации	18
3.2 Реализация загружаемого модуля ядра	18
3.2.1 Функции загрузки и выгрузки модуля	18
3.2.2 Функции для взаимодействия с файловой системой	20
3.2.3 Функции для получения информации о процессах	20

3.3	Пользовательское приложение . . . . .	20
3.3.1	Взаимодействие с модулем ядра . . . . .	20
3.3.2	Получение информации о системе . . . . .	22
4	Исследовательская часть . . . . .	24
	ЗАКЛЮЧЕНИЕ . . . . .	25
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . . .	27
	ПРИЛОЖЕНИЕ А . . . . .	28
	ПРИЛОЖЕНИЕ Б . . . . .	28
	ПРИЛОЖЕНИЕ В . . . . .	29
	ПРИЛОЖЕНИЕ Г . . . . .	30
	ПРИЛОЖЕНИЕ Д . . . . .	31
	ПРИЛОЖЕНИЕ Е . . . . .	33

## ВВЕДЕНИЕ

Процесс является важной частью операционной системы Linux. Поскольку его роль крайне высока, нередко существует необходимость получения различной информации о нем. Например, страницы занимаемой памяти или файлы, которые открыты процессом. При этом важно иметь возможность получить интересующие данные для каждого процесса и в простом доступе. Операционная система Linux позволяет получить доступ к данным благодаря тому, что ее ядро имеет открытый исходный код.

В рамках курсовой работы поставлена цель — разработать загружаемый модуль ядра для получения информации о процессах, их дереве, а также пользовательское приложение для вывода полученной информации от модуля и об операционной системе Linux в целом.

# 1 Аналитическая часть

## 1.1 Постановка задачи

В соответствии с заданием необходимо разработать загружаемый модуль ядра, который позволит получить следующую информацию:

- дерево процессов системы;
- открытые процессом файлы (их имя и размер);
- используемую процессом память (адреса и размер).

Также в рамках задания необходимо разработать пользовательское приложение для вывода информации из загружаемого модуля, а также получить следующую важную информацию об операционной системе Linux:

- имя компьютера;
- имя пользователя;
- версию ядра операционной системы;
- время непрерывной работы системы;
- модель процессора и его частота;
- модель видеокарты;
- загруженность процессора;
- общее и используемое количество ОЗУ.

## 1.2 Способы получения информации о системе

Для того, чтобы в Linux получить информацию о системе в режиме пользователя, существуют следующие подходы.

1. **Команда dmesg.** Используется для вывода кольцевого буфера ядра. Дает возможность получить сообщения от процессов, которые имеют доступ к ядру. К данной команде применима опция **grep** для вывода только интересующей части информации.

2. **Программа fdisk.** Является инструментом для работы с таблицей разбиения диска. Физические диски обычно разбиваются на несколько логических дисков, которые называются разделами диска. Информация о разбиении физического диска на разделы хранится в таблице разбиения диска, которая находится в нулевом секторе физического диска. Если имеется два или более дисков, и необходимо получить данные о конкретном диске, нужно указать в команде желаемый диск, например `fdisk -l /dev/<имя>`.
3. **Утилита dmidecode.** Данная утилита выводит содержимое таблицы DMI (Desktop Management Interface) системы в формате, предназначенном для восприятия человеком. Эта таблица содержит информацию, относящуюся к компонентам аппаратного обеспечения системы, а также сведения о версии BIOS и так далее. В выводе `dmidecode` не только содержится описание текущей конфигурации системы, но и приводятся данные о максимально допустимых значениях параметров, например, о поддерживаемых частотах работы CPU, максимально возможном объеме памяти и так далее.
4. **Каталог /proc.** Каталог `/proc` содержит директории и файлы, которые содержат информацию о системе и ее процессах. Из них можно получить информацию об оперативной памяти или центральном процессоре в реальном времени. Поскольку данные хранятся в файлах, то они должны быть прочитаны, а в некоторых случаях, верно интерпретированы пользователем, чтобы получить нужную информацию.

Таким образом, в качестве способа получения информации о системе будет выбран каталог `/proc`. Он позволяет получить все данные в соответствии с заданием, а также для доступа к файлам не требуются права суперпользователя. То есть информация может быть получена не только администратором компьютера, но и обычным пользователем.

### 1.3 Получение информации о системе

Каждый файл в каталоге `/proc` хранит определенные данные о системе. Таким образом, для получения необходимой по заданию информации нужно обратиться к следующим файлам каталога `/proc`.

### 1.3.1 Имя компьютера

Чтобы узнать собственное имя компьютера с помощью `procfs`, независимое от сетевых интерфейсов, необходимо прочитать информацию из файла `/proc/sys/kernel/hostname`.

### 1.3.2 Информация о процессоре

Для получения модели и частоты центрального процессора, необходимо проанализировать файл `/proc/cpuinfo`, в котором хранится информация о процессоре и его состоянии в реальном времени. Модель процессора считывается из поля `model name`, а частота из поля `cpu MHz`.

### 1.3.3 Информация о видеокарте

На компьютере, на котором выполнялась курсовая работа, установлена видеокарта от компании `Nvidia`. По этой причине информацию о видеокарте можно получить из следующего файла в файловой подсистеме `procfs`: `/proc/driver/nvidia/gpus/0000:01:00.0/information`.

### 1.3.4 Версия ядра операционной системы

Информация о версии ядра операционной системы `Linux` доступна в файле `/proc/version`. При этом, там также хранится название ОС, версия и название установленного дистрибутива.

### 1.3.5 Загруженность процессора

Для вычисления процента загруженности центрального процессора, следует проанализировать файл `/proc/stat`, в котором находится информация об активности процессора. Необходимая информация хранится в первых четырех полях строки `cpu`:

- число процессов, выполняющихся в режиме пользователя;
- число процессов с изменённым приоритетом (`nice`), выполняющихся в режиме пользователя;
- число процессов, выполняющихся в режиме ядра;

— число процессов, выполняющих функцию простоя процессора (`idle`).

### 1.3.6 Информация об оперативной памяти компьютера

Информация об оперативной памяти может быть получена путем анализа файла `/proc/meminfo`. Объем всей оперативной памяти считывается из поля `MemTotal`, а объем памяти, доступной для немедленного её выделения процессам из поля `MemAvailable`. Таким образом, объем используемой оперативной памяти вычисляется из разности этих значений.

## 1.4 Загружаемый модуль ядра

Ядро Linux относится к категории монолитных ядер — это означает, что большая часть функциональности операционной системы реализована ядром и запускается в привилегированном режиме. Этот подход отличен от подхода микроядра, когда в режиме ядра выполняется только основная функциональность (взаимодействие между процессами, диспетчеризация, базовый ввод-вывод, управление памятью), а остальная функциональность вытесняется за пределы привилегированной зоны (драйверы, сетевой стек, файловые системы).

Ядро Linux динамически изменяемое. Это означает, что можно загружать в ядро дополнительную функциональность, выгружать функции из ядра и даже добавлять новые модули, использующие другие модули ядра. Преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули.

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (`main`) в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии ядра 2.6 эти функции можно именовать как угодно).

Функция входа загружаемого модуля ядра вызывается, когда модуль загружается в ядро, а функция выхода — соответственно при выгрузке из ядра. Поскольку функции входа и выхода определяются программистом, для указания назначения этих функций используются макросы `module_init` и `module_exit`.



Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание модуля, а также другие параметры.

### 1.4.1 Загрузка и выгрузка загружаемого модуля ядра

Процесс загрузки модуля начинается в пользовательском пространстве с команды `insmod`. Команда `insmod` определяет модуль для загрузки и выполняет системный вызов уровня пользователя `init_module` для начала процесса загрузки.

Функция `init_module` работает на уровне системных вызовов и вызывает функцию ядра `sys_init_module`. Это основная функция для загрузки модуля, обращающаяся к нескольким другим функциям для решения специальных задач. Аналогичным образом команда `rmmod` выполняет системный вызов функции `delete_module`, которая обращается в ядро с вызовом `sys_delete_module` для удаления модуля из ядра.

## 1.5 Взаимодействие пространства ядра и пользователя

В Linux для передачи данных из пространства ядра в пространство пользователя зачастую используется виртуальная файловая система `procfs`, которая предоставляет системные вызовы для реализации интерфейса между двумя этими пространствами.

Для работы в системе используется структура `proc_ops` определена в файле `linux/proc_fs.h` и содержит в себе указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством. Основные поля структуры представлены в листинге 1.1.

Листинг 1.1 — Структура `proc_ops`

```
1  struct proc_ops {
2      unsigned int proc_flags;
3      int          (*proc_open)(struct inode *, struct file *);
4      ssize_t      (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5      ssize_t      (*proc_read_iter)(struct kiocb *, struct iov_iter *);
6      ssize_t      (*proc_write)(struct file *, const char __user *, size_t,
   ↪ loff_t *);
```

```

7      /* mandatory unless nonseekable_open() or equivalent is used */
8      loff_t      (*proc_lseek)(struct file *, loff_t, int);
9      int         (*proc_release)(struct inode *, struct file *);
10     __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
11     long        (*proc_ioctl)(struct file *, unsigned int, unsigned long);
12 } __randomize_layout;

```

Также для передачи данных между пространством пользователя и ядра используются две функции – `copy_to_user` (позволяет копировать блоки данных из пространства ядра в пространство пользователя) и `copy_from_user` (позволяет копировать блоки данных из пространства пользователя в пространство ядра). Определены эти функции в файле `linux/uaccess.h` и возвращают количество байт, которые не удалось записать.

Для приема управляющих данных и передачи результатов функций в пользовательское приложение целесообразным будет создать три файла в каталоге `/proc`, каждый из которых будет являться интерфейсом соответствующей функции разрабатываемого модуля. Интерфейс для взаимодействия с пространством пользователя представляется с помощью `struct proc_dir_entry`. Некоторые поля данной структуры представлены в приложении 4.

## 1.6 Получение информации о процессах

Процесс — это программа в стадии выполнения. Он состоит из исполняемого программного кода, набора ресурсов (таких, как открытые файлы), внутренних данных ядра, адресного пространства, одного или нескольких потоков исполнения и секции данных, содержащей глобальные переменные.

С каждым процессом связан (ассоциирован) «описатель процесса» или дескриптор процесса. Ядро Linux использует циклически замкнутый двусвязный список записей `struct task_struct` (приложение 4) для хранения дескрипторов процессов. Эта структура объявлена в файле `linux/sched.h`. Дескриптор содержит информацию, используемую для того, чтобы отслеживать процесс в оперативной памяти. В частности, в дескрипторе содержатся идентификатор процесса (PID), его состояние, ссылки на родительский и дочерние процессы, регистры процессора, список открытых файлов и информация об адресном пространстве.

### 1.6.1 Предоставление процессов в памяти

Строка `struct list_head tasks` внутри определения `struct task_struct` показывает, что ядро использует циклический связанный список для хранения задач. Это означает, что можно использовать стандартные макросы и функции для работы со связанными списками с целью просмотра полного списка задач.

«Родителем всех процессов» в системе Linux является процесс `init`. Так что он должен стоять в начале списка, хотя, строго говоря, начала не существует раз речь идет о циклическом списке. Дескриптор процесса `init` задается статично, о чем говорит следующая строчка в структуре `extern struct task_struct init_task`.

Имеется несколько макросов и функций, которые помогают перемещаться по этому списку:

- `for_each_process()` — это макрос, который проходит весь список задач;
- `next_task()` — макрос, определенный в `linux/sched.h`, возвращает следующую задачу из списка.

### 1.6.2 Построение дерева процессов

Построение дерева процессов основано на иерархии «предок-потомок». Таким образом, для корректного построения дерева необходимо рекурсивно пройти по всем потомкам процесса, начиная с `init`. При этом необходимо запоминать глубину рекурсии на каждом шаге для корректного отображения дерева.

### 1.6.3 Получение информации о памяти процесса

Адресное пространство процесса представлено полем `mm` типа `struct mm_struct`. Для получения необходимой информации в рамках задачи интерес представляет поле `mmap` данной структуры, которое имеет тип `struct vm_area_struct`. Данное поле представляет собой список областей памяти процесса.

Данная структура определена в файле `linux/mm_types.h`, некоторые ее поля представлены в приложении 4. Особый интерес представляет поле

`vm_next`, которое указывает на следующую область памяти, а также поля `vm_start` и `vm_end`, с помощью которых можно получить информацию об адресах начала и конца области памяти, а также вычислить ее размер.

#### 1.6.4 Получение информации об открытых процессах файлах

Для получения информации о файлах, открытых процессом, используется структура `struct fdtable`, определенная в файле `linux/fdtable.h` (приложение 4). Данная структура имеет поле `fd`, которое определяет массива указателей на дескрипторы файлов, открытых процессом. Таким образом, обратившись к полям каждого элемента массива `fd`, можно получить необходимую информацию: размер файла, путь, имя.

### Вывод

В данном разделе были рассмотрены некоторые функции и структуры операционной системы Linux, которые будут использованы при написании программного продукта. Также были описаны такие понятия системы, как загружаемый модуль ядра, процесс, файловая система `procfs`.

## 2 Конструкторская часть

В состав разработанного программного обеспечения входит один загружаемый модуль ядра, который записывает в файлы файловой системы `/proc` информацию об открытых файлах процесса, о памяти, выделенной под него, а также дерево всех процессов системы. Также разработано пользовательское приложение, которое считывает данные от загружаемого модуля, и выводит общую информацию о системе.

### 2.1 Диграмма состояний модуля ядра

На рисунке 2.1 представлена диаграмма IDEF0 нулевого уровня, а на рисунке 2.2 — первого уровня.



Рисунок 2.1 – Диаграмма IDEF0 нулевого уровня

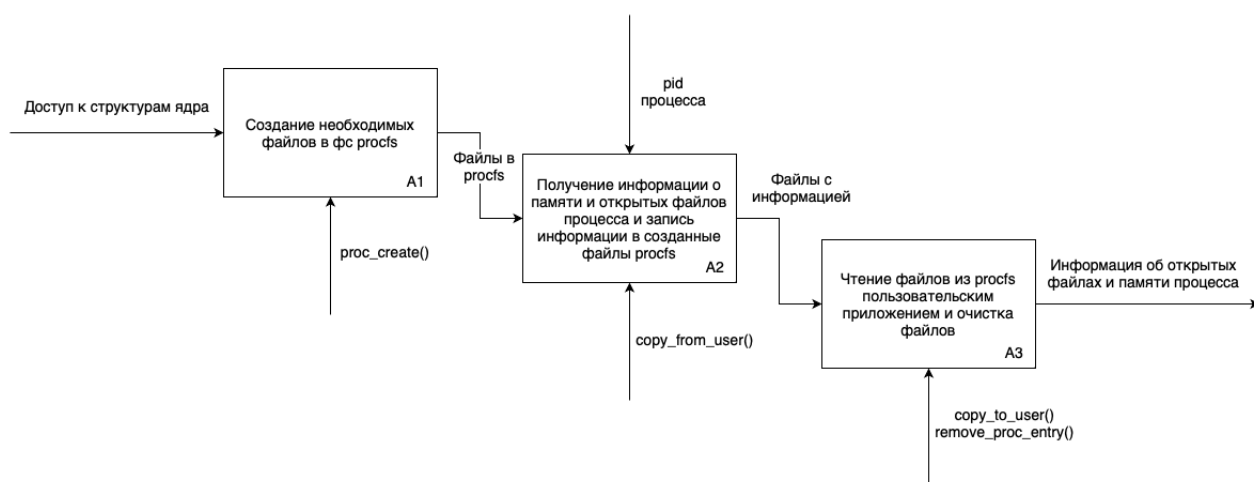


Рисунок 2.2 – Диаграмма IDEF0 первого уровня

## 2.2 Структура программного обеспечения

На рисунке 2.3 представлена структура программного обеспечения.

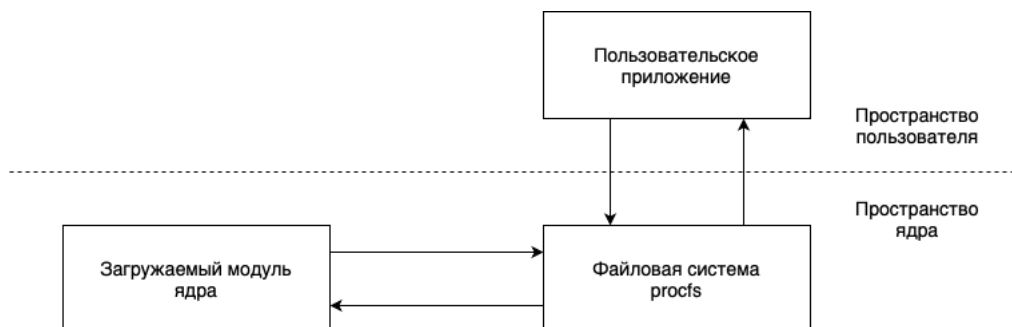


Рисунок 2.3 – Структура программного обеспечения

## 2.3 Схемы алгоритмов функций загружаемого модуля ядра

На рисунке 2.4 представлена схема алгоритма функции получения списка открытых процессом файлов, а на рисунке 2.5 — схема алгоритма функции получения информации о памяти, выделенной процессу.

## 2.4 Пользовательское приложение

Пользовательское приложение предоставляет пользователю информацию о системе, обновляющуюся каждую секунду. Также имеет функции для получения информации, формируемой загружаемым модулем ядра. Процесс получения информации из пространства ядра состоит из двух этапов:

- передача управляющей информации из пространства пользователя в пространство ядра;
- чтение из пространства ядра информации, полученной в результате работы функций модуля ядра.

## Вывод

В данном разделе были приведена диаграмма состояний модуля ядра, а также структура программного обеспечения. При этом выделены важные функции загружаемого модуля ядра — функция получения списка открытых процессом файлов и функция получения информации о памяти, выделенной процессу.

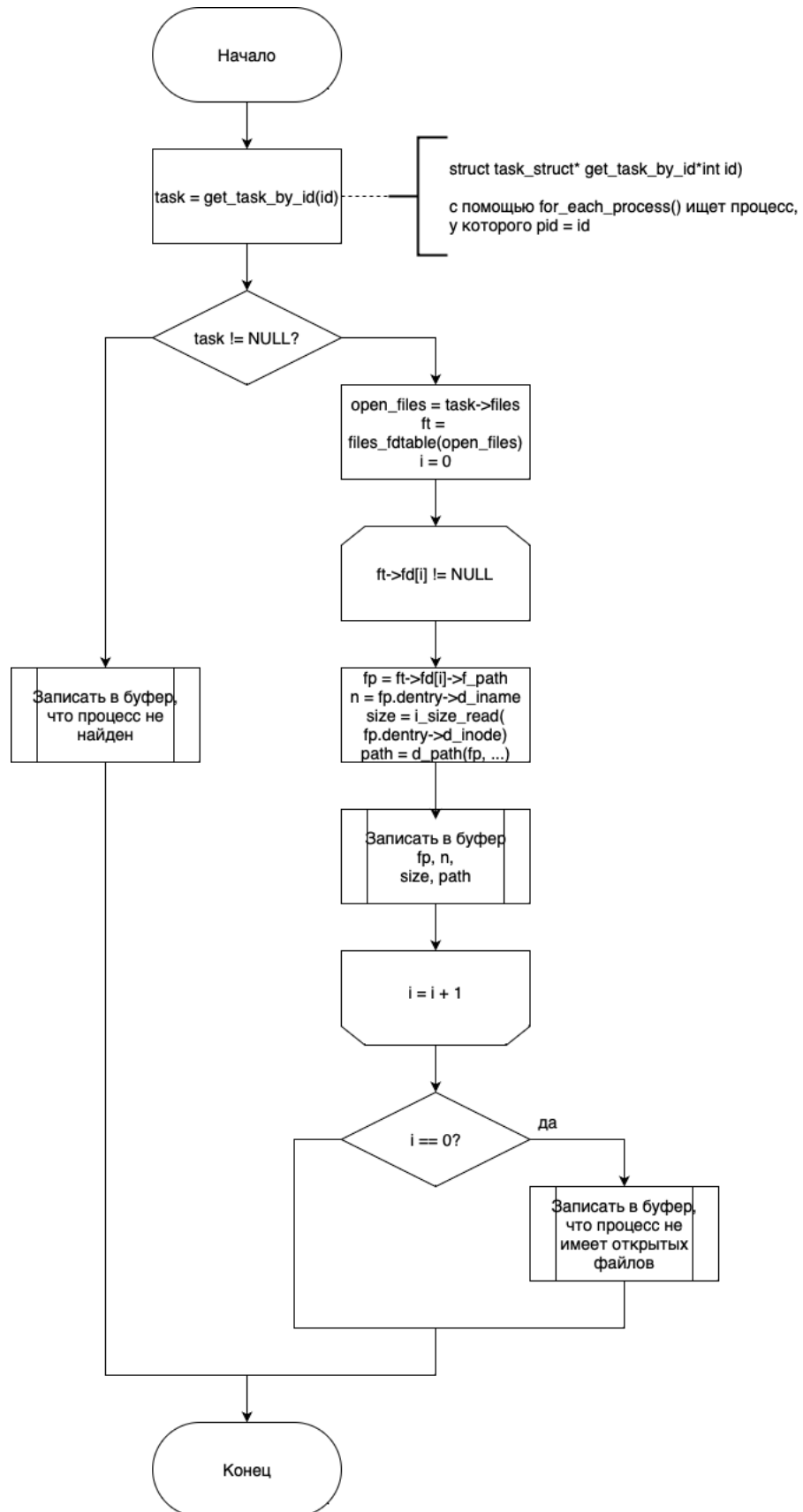


Рисунок 2.4 – Функция получения списка открытых процессом файлов

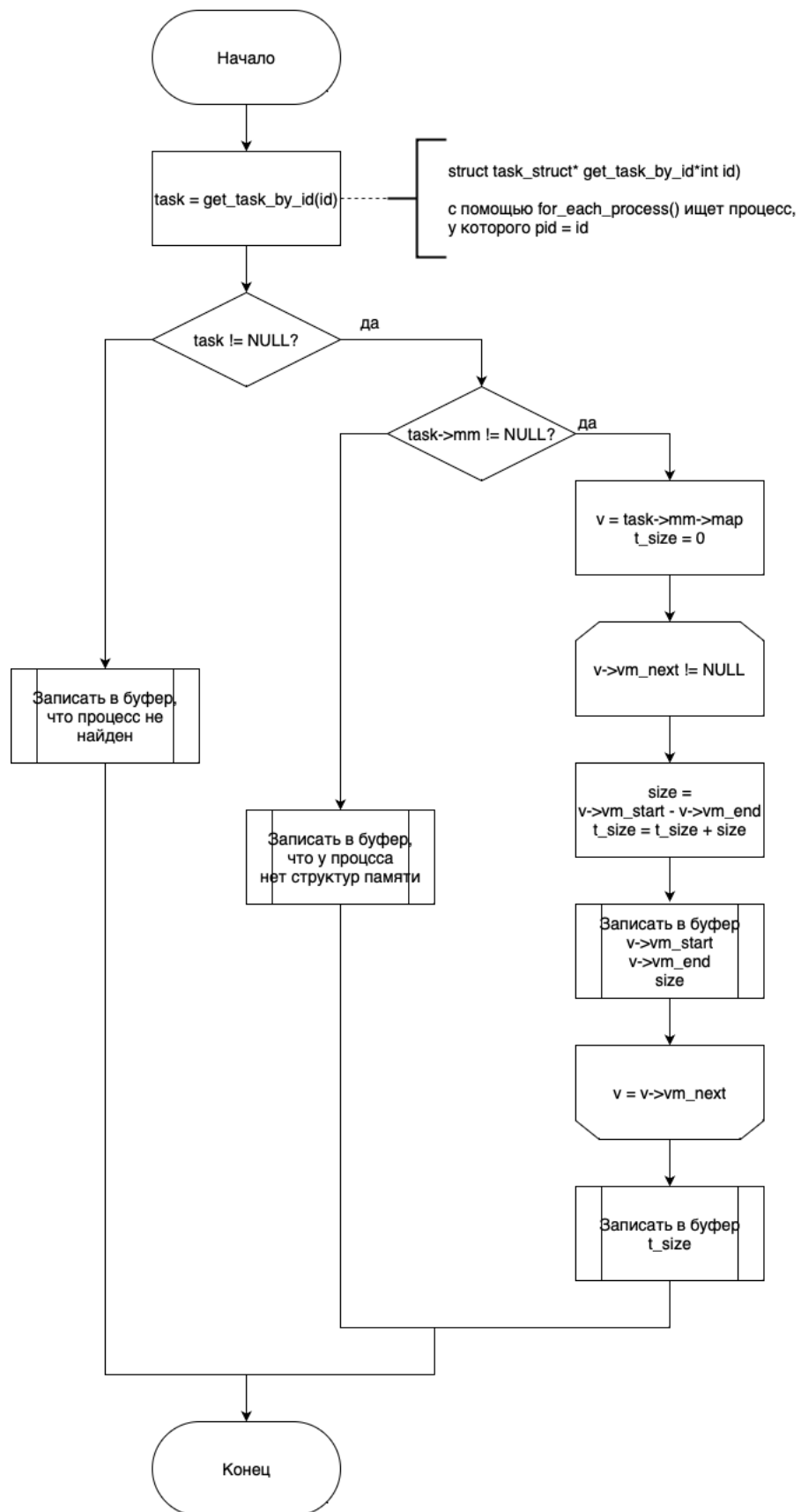


Рисунок 2.5 – Структура получения информации о памяти, выделенной процессу



## 3 Технологическая часть

### 3.1 Средства реализации

В качестве языка программирования для написания загружаемого модуля ядра был выбран **C**. Решение обусловлено тем, что сама операционная система **Linux** написана на языке **C**, поэтому все структуры и функции также написаны на нем.

Для написания пользовательского приложения был выбран язык **C++**, так как он обладает широким набором функций, а также графической библиотекой **Qt**.

Реализация была написана на дистрибутиве **Ubuntu 22.04**, который работает на операционной системе **Linux** версии **5.15.0-58**.

### 3.2 Реализация загружаемого модуля ядра

В листинге 3.1 представлено содержание **Makefile** для сборки загружаемого модуля. Также в нем содержится цель **clean** для очистки файлов, которые создаются при сборке модуля.

Листинг 3.1 — Makefile для сборки загружаемого модуля ядра

```
1 EXTRA_CFLAGS += -std=gnu99
2 obj-m += mymodule.o
3 mymodule-objs := func.o mod_proc.o
4
5 all:
6     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7
8 clean:
9     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

#### 3.2.1 Функции загрузки и выгрузки модуля

В листинге 3.2 представлена функция, которая вызывается при загрузке модуля в ядро, а в листинге 3.3 — при выгрузке модуля.

Листинг 3.2 — Функция для загрузки модуля

```
1 static int __init proc_init( void )
2 {
3     printk("Start module\n");
```

## Продолжение листинга 3.2

```
4     static struct proc_dir_entry *procfs_tree;
5     static struct proc_dir_entry *procfs_mem;
6     static struct proc_dir_entry *procfs_files;
7
8     static struct proc_ops proc_file_fops_tree = {
9         .proc_read = read_proc_tree,
10        .proc_write = write_proc_tree
11    };
12
13    static struct proc_ops proc_file_fops_mem = {
14        .proc_read = read_proc_mem,
15        .proc_write = write_proc_mem
16    };
17
18    static struct proc_ops proc_file_fops_files = {
19        .proc_read = read_proc_files,
20        .proc_write = write_proc_files
21    };
22
23    procfs_tree = proc_create(PROCFS_TREE_NAME, S_IFREG | S_IRUGO | S_IWUGO, NULL,
↪ &proc_file_fops_tree);
24    procfs_mem = proc_create(PROCFS_MEM_NAME, S_IFREG | S_IRUGO | S_IWUGO, NULL,
↪ &proc_file_fops_mem);
25    procfs_files = proc_create(PROCFS_FILES_NAME, S_IFREG | S_IRUGO | S_IWUGO, NULL,
↪ &proc_file_fops_files);
26
27    msg_tree = kmalloc(MAX_WRITE_BUF_SIZE * sizeof(char), GFP_KERNEL);
28    msg_mem = kmalloc(MAX_WRITE_BUF_SIZE * sizeof(char), GFP_KERNEL);
29    msg_files = kmalloc(MAX_WRITE_BUF_SIZE * sizeof(char), GFP_KERNEL);
30
31    if (procfs_tree == NULL || procfs_mem == NULL || procfs_files == NULL)
32    {
33        remove_proc_entry(PROCFS_TREE_NAME, NULL);
34        remove_proc_entry(PROCFS_MEM_NAME, NULL);
35        remove_proc_entry(PROCFS_FILES_NAME, NULL);
36        printk("Error: Could not initialize files in /proc\n");
37        return -ENOMEM;
38    }
39
40    return 0;
41 }
```

## Листинг 3.3 — Функция для выгрузки модуля

```
1     static void __exit proc_exit( void )
2     {
3         printk("Procfs cleanup\n");
```

## Продолжение листинга 3.3

```
4     kfree(msg_tree);
5     kfree(msg_mem);
6     kfree(msg_files);
7     remove_proc_entry(PROCFS_TREE_NAME, NULL);
8     remove_proc_entry(PROCFS_MEM_NAME, NULL);
9     remove_proc_entry(PROCFS_FILES_NAME, NULL);
10    printk("Success cleanup module\n");
11 }
```

### 3.2.2 Функции для взаимодействия с файловой системой

Для взаимодействия с `procfs` необходимо было написать функции для чтения и записи. В приложении 4 приведен листинг функций `read_proc_mem()` и `write_proc_mem()`.

### 3.2.3 Функции для получения информации о процессах

В приложении 4 представлены листинги функций:

- построения дерева процессов;
- получения информации об используемой процессом памяти;
- получения информации об открытых процессом файлах.

## 3.3 Пользовательское приложение

Собирается проект с помощью автосгенерированного `Makefile` по проекту в `Qt`.

### 3.3.1 Взаимодействие с модулем ядра

В листинге 3.4 представлена функция для «подключения» (проверки того, что он активен и создал файлы в `procfs`) к загружаемому модулю ядра. А в листинге 3.5 функция для получения информации из `procfs` от загружаемого модуля (приведен пример для получения информации о памяти процесса).

### Листинг 3.4 — Функция для «подключения» к модулю ядра

```
1  int MainWindow::connect_module(const std::string& str_write, const std::string&
   ↪ filename, std::vector<std::string>& answer)
2  {
3      std::ofstream fw;
4      fw.open(filename);
5
6      if(fw.is_open())
7      {
8          fw << str_write;
9          qDebug() << "Успешно записано " << QString(str_write.data()) << " в " <<
   ↪ QString(filename.data());
10         fw.close();
11     }
12     else
13         return -1;
14
15     std::ifstream fr(filename);
16
17     if(fr.is_open())
18     {
19         std::string line;
20         bool eof = fr.eof();
21
22         if (eof)
23         {
24             answer.push_back("No such ID!");
25             return 0;
26         }
27
28         while (std::getline(fr, line) && line.length() != 0)
29         {
30             qDebug() << QString(line.data());
31             answer.push_back(line);
32         }
33
34         fr.close();
35         return 0;
36     }
37
38     return -1;
39 }
```

Листинг 3.5 — Функция для получения информации о памяти процесса, полученной от загружаемого модуля

```
1 void MainWindow::on_btn_mem_clicked()
2 {
3     std::vector<std::string> answer;
4     std::string filename("/proc/mod_proc_mem");
5     std::string str_write(std::to_string(ui->spb_mem->value()));
6
7     int res = this->connect_module(str_write, filename, answer);
8
9     if (res == 0)
10    {
11        ui->tb_mem->clear();
12        for(std::string s : answer)
13        {
14            ui->tb_mem->append(QString(s.data()));
15        }
16    }
17    else
18    {
19        QMessageBox::warning(this, "Модуль не доступен",
20            "Невозможно обратиться к модулю.\nВозможно, он не загружен\nили
↪ используется другим процессом");
21    }
22 }
```

### 3.3.2 Получение информации о системе

В листинге 3.6 приведен пример получения информации из системы, путем прочтения файла из каталога `/proc` (пример получения информации о модели процессора).

Листинг 3.6 — Функция для получения модели процессора

```
1 QString get_cpu_model()
2 {
3     std::ifstream stream("/proc/cpuinfo");
4     std::string str;
5
6     for(int i = 0; i < 16; i++)
7         stream >> str;
8
9     getline(stream, str);
10
11     stream.close();
```

### Продолжение листинга 3.6

```
12     return QString::fromStdString(str);  
13 }
```

## Вывод

В данной разделе приведены средства реализации (для модуля язык С, для пользовательского приложения язык С++). Также представлены листинги основных функций программного продукта.

## 4 Исследовательская часть

...

## ЗАКЛЮЧЕНИЕ

...



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Lerner A.* Analyzing the Use of Quick Response Codes in the Wild // Греция, Conference: the 13th Annual International Conference. — 2020.
2. Штрих код — назначение и использование [Электронный ресурс]. — Режим доступа URL: <https://rossertcentr.ru/shtrihkod-naznachenie-i-ispolzovanie> (Дата обращения: 05.11.2022).
3. 2D Barcode Reader User Guide [Электронный ресурс]. — Режим доступа URL: [https://www.perco.ru/download/documentation/rus/Mertech\\_N200\\_P2D\\_ApplicGuide.pdf](https://www.perco.ru/download/documentation/rus/Mertech_N200_P2D_ApplicGuide.pdf) (Дата обращения: 15.11.2022).
4. GS1 — Standards and services [Электронный ресурс]. — Режим доступа URL: <https://www.gs1.org/standards/barcodes> (Дата обращения: 5.05.2022).
5. *Воронков А.* Технологии штрихового кодирования в логистике // Россия, Нижний Новгород, ННГАСУ. — 2011.
6. Регистрация штриховых кодов EAN-13 [Электронный ресурс]. — Режим доступа URL: <https://www.ean-13.ru/index.html> (Дата обращения: 18.11.2022).
7. Спецификация символики штрихового кода Code 128 [Электронный ресурс]. — Режим доступа URL: <https://docs.cntd.ru/document/1200105469> (Дата обращения: 16.11.2022).
8. Code 39 bar code symbology specification [Электронный ресурс]. — Режим доступа URL: <https://www.iso.org/standard/43897.html> (Дата обращения: 16.11.2022).
9. *Ahmad A.* Significance of the UPC Barcode in the Food Industry // Баку, Baku Scientific Researches Conference. — 2021.
10. Спецификация символики штрихового кода Interleaved 2 of 5 [Электронный ресурс]. — Режим доступа URL: <https://docs.cntd.ru/document/1200158535> (Дата обращения: 20.11.2022).
11. PDF417 bar code symbology specification [Электронный ресурс]. — Режим доступа URL: <https://www.iso.org/standard/65502.html> (Дата обращения: 20.11.2022).

12. Спецификация символики Data Matrix [Электронный ресурс]. — Режим доступа URL: <https://docs.cntd.ru/document/1200071931> (Дата обращения: 20.11.2022).
13. Спецификация символики штрихового кода Aztec Code [Электронный ресурс]. — Режим доступа URL: <https://docs.cntd.ru/document/1200079075> (Дата обращения: 20.11.2022).
14. Спецификация символики штрихового кода QR Code [Электронный ресурс]. — Режим доступа URL: <https://docs.cntd.ru/document/1200121043> (Дата обращения: 20.11.2022).
15. *Fritz J.* Data Matrix and PDF 417 data integrity test // Огайо, Ohio University, Center for Automatic Identification. — 2020.
16. The advantages and disadvantages of barcodes and radio frequency identification in supply chain management [Электронный ресурс]. — Режим доступа URL: <https://ro.uow.edu.au/cgi/viewcontent.cgi?article=1009&context=thesesinfo> (Дата обращения: 20.11.2022).

## ПРИЛОЖЕНИЕ А

Листинг 4.1 — Структура task\_struct

```
1  struct task_struct {
2      volatile long state;
3      void *stack;
4      unsigned int flags;
5      ...
6      int prio, static_prio;
7      ...
8      struct list_head tasks;
9
10     struct mm_struct *mm, *active_mm;
11     ...
12     pid_t pid;
13     pid_t tgid;
14     ...
15     struct task_struct *real_parent;
16     ...
17     char comm[TASK_COMM_LEN];
18     ...
19     struct thread_struct thread;
20     ...
21     struct files_struct *files;
22     ...
23 }
```

## ПРИЛОЖЕНИЕ Б

Листинг 4.2 — Структура `vm_area_struct`

```
1  struct vm_area_struct
2  {
3      struct mm_struct * vm_mm; /* параметры области виртуальной памяти */
4      unsigned long vm_start;
5      unsigned long vm_end;
6      /* Связанный список областей задачи отсортированный по адресам */
7      struct vm_area_struct *vm_next;
8      ...
9      unsigned short vm_flags;
10     ...
11     struct vm_operations_struct * vm_ops; /*операции над областью */
12     ...
13     unsigned long vm_pte; /* разделяемая память */
14 };
```

## ПРИЛОЖЕНИЕ В

Листинг 4.3 — Структура fdtable

```
1  struct fdtable
2  {
3      unsigned int max_fds;
4      struct file **fd;
5      unsigned long *close_on_exec;
6      unsigned long *open_fds;
7      unsigned long *full_fds_bits;
8      struct rcu_head rcu;
9  };
```

## ПРИЛОЖЕНИЕ Г

Листинг 4.4 — Структура `proc_dir_entry`

```
1  struct proc_dir_entry
2  {
3      ...
4      const char *name;
5      mode_t mode;
6      ...
7      uid_t uid;
8      gid_t gid;
9      ...
10     const struct file_operations *proc_fops;
11     ...
12     read_proc_t *read_proc;
13     write_proc_t *write_proc;
14     ...
15 }
```

## ПРИЛОЖЕНИЕ Д

Листинг 4.5 — Функции записи и чтения для структуры `file_operations`  
получения памяти процесса

```
1  static ssize_t read_proc_mem(struct file *filp, char *buf, size_t count, loff_t *offp )
2  {
3      char msg2[] = "Pass id of process";
4      char *buf_msg = vmalloc(1000000 * sizeof(char));
5
6      if (is_number(msg_mem, strlen(msg_mem)))
7      {
8          int id = str_to_number(msg_mem, strlen(msg_mem));
9          mem_info(id, buf_msg);
10     }
11     else
12     {
13         memcpy(buf_msg, msg2, strlen(msg2));
14         buf_msg[strlen(msg2)] = '\0';
15     }
16
17     int res;
18
19     if(*offp >= strlen(buf_msg))
20     {
21         *offp = 0;
22         return 0;
23     }
24
25     if(count > strlen(buf_msg) - *offp)
26         count = strlen(buf_msg) - *offp;
27
28     res = copy_to_user((void*)buf, buf_msg + *offp, count);
29
30     *offp += count;
31     return count;
32 }
33
34 static ssize_t write_proc_mem(struct file *filp, const char *buf, size_t count, loff_t
↵ *offp)
35 {
36     ssize_t procfs_buf_size = count;
37
38     if (procfs_buf_size > MAX_WRITE_BUF_SIZE)
39     {
40         procfs_buf_size = MAX_WRITE_BUF_SIZE;
41     }
```

## Продолжение листинга 4.5

```
42
43     copy_from_user(msg_mem, buf, procfs_buf_size);
44
45     msg_mem[procfs_buf_size] = '\0';
46     printk("Message to /proc/%s from user-space: %s\n", PROCFS_MEM_NAME, msg_mem);
47
48     return count;
49 }
```



## ПРИЛОЖЕНИЕ Е

Листинг 4.6 — Функция получения информации о памяти

```
1  void mem_info(int id, char* buf)
2  {
3      char *str = vmalloc(10000 * sizeof(char));
4      int offset = 0;
5      int cnt;
6      bool found = false;
7
8      struct task_struct *current_task = current;
9      struct vm_area_struct *v;
10     unsigned long t_size = 0;
11
12     struct task_struct *task = get_task_by_id(id);
13
14     if (task)
15     {
16         found = true;
17         current_task = task;
18     }
19
20     if (found && current_task->mm != NULL)
21     {
22         printk("====VIRTUAL MEMORY INFORMATION====\n\n");
23         v = current_task->mm->mmap;
24
25         if(v != NULL)
26         {
27             printk("Process: %s[%d]\n", current_task->comm, current_task->pid);
28             cnt = sprintf(str, "Process: %s[%d]\n", current_task->comm,
↪ current_task->pid);
29
30             str[cnt] = '\0';
31             memcpy(buf + offset, str, strlen(str));
32             offset += strlen(str);
33
34             int is_full = 0;
35
36             while(v->vm_next != NULL)
37             {
38                 unsigned long size = v->vm_end - v->vm_start;
39
40                 if (size > 100000)
41                 {
42                     printk("Full house\n");
```

## Продолжение листинга 4.6

```
43         is_full = 1;
44         break;
45     }
46
47     t_size = t_size + size;
48
49     printk("Start: 0x%lx, End: \t0x%lx \t(block Size: \t0x%lx)\n",
↪ v->vm_start, v->vm_end, size);
50     cnt = sprintf(str, "Start: \t0x%lx, \tEnd: \t0x%lx \t(block Size:
↪ \t0x%lx)\n", v->vm_start, v->vm_end, size);
51
52     str[cnt] = '\0';
53     memcpy(buf + offset, str, strlen(str));
54     offset += strlen(str);
55     v = v->vm_next;
56 }
57
58 if (is_full)
59 { }
60 else
61 {
62     printk("Total size of virtual space is: 0x%lx\n", t_size);
63     cnt = sprintf(str, "Total size of virtual space is: 0x%lx\n", t_size);
64
65     str[cnt] = '\0';
66     memcpy(buf + offset, str, strlen(str));
67     offset += strlen(str);
68 }
69 }
70
71 else if (current_task->mm == NULL)
72 {
73     printk("ID %d have no memory structure.\n", id);
74     cnt = sprintf(str, "ID %d have no memory structure.\n", id);
75
76     str[cnt] = '\0';
77     memcpy(buf + offset, str, strlen(str));
78     offset += strlen(str);
79 }
80 else
81 {
82     printk("ID %d not found\n", id);
83     cnt = sprintf(str, "ID %d not found\n", id);
84
85     str[cnt] = '\0';
86     memcpy(buf + offset, str, strlen(str));
87     offset += strlen(str);
```

## Продолжение листинга 4.6

```
88     }
89
90     buf[offset] = '\0';
91     vfree(str);
92 }
```

## Листинг 4.7 — Функция получения информации об открытых процессом файлах

```
1  void files_info(int id, char* buf)
2  {
3      char *str = vmalloc(1000 * sizeof(char));
4      int offset = 0;
5      int cnt;
6
7      bool found = false;
8      struct task_struct *current_task = current;
9      struct files_struct *open_files;
10     struct fdtable *files_table;
11     struct path files_path;
12
13     struct task_struct *task = get_task_by_id(id);
14
15     if (task)
16     {
17         found = true;
18         current_task = task;
19     }
20
21     if (found)
22     {
23         printk("====OPEN FILES INFORMATION====\n\n");
24         printk("Process: %20s[%4d]\n", current_task->comm, current_task->pid);
25         cnt = sprintf(str, "Process: %20s[%4d]\n", current_task->comm,
↪ current_task->pid);
26
27         str[cnt] = '\0';
28         memcpy(buf + offset, str, strlen(str));
29         offset += strlen(str);
30
31         int i = 0;
32         open_files = current_task->files;
33         files_table = files_fdtable(open_files);
34         char *path;
35         char *buf_tmp = (char*)kmallocc(10000 * sizeof(char), GFP_KERNEL);
36
37         while(files_table->fd[i] != NULL)
```

## Продолжение листинга 4.7

```
38     {
39         files_path = files_table->fd[i]->f_path;
40         char* name = files_table->fd[i]->f_path.dentry->d_iname;
41         long long size = i_size_read(files_table->fd[i]->f_path.dentry->d_inode);
42         path = d_path(&files_path, buf_tmp, 10000 * sizeof(char));
43
44         printk("Name: %s, %tFD: %t%d, %tSize: %t0x%llx bytes %t(path: %t%s)\n",
↪ name, i, size , path);
45         cnt = sprintf(str, "Name: %s, %tFD: %t%d, %tSize: %t0x%llx bytes
↪ %t(path: %t%s)\n", name, i, size , path);
46
47         str[cnt] = '\0';
48         memcpy(buf + offset, str, strlen(str));
49         offset += strlen(str);
50         i++;
51     }
52
53     if (i == 0) {
54         printk("Process hasn't opened files");
55         cnt = sprintf(str, "Process hasn't opened files.");
56
57         str[cnt] = '\0';
58         memcpy(buf + offset, str, strlen(str));
59         offset += strlen(str);
60     }
61     kfree(buf_tmp);
62 }
63 else
64 {
65     printk("ID %d not found\n", id);
66     cnt = sprintf(str, "ID %d not found\n", id);
67
68     str[cnt] = '\0';
69     memcpy(buf + offset, str, strlen(str));
70     offset += strlen(str);
71 }
72
73 buf[offset] = '\0';
74 vfree(str);
75 }
```

## Листинг 4.8 — Функции получения дерева процессов в системе

```
1 void process_info(struct task_struct* task, int n, char* buf, int* offset)
2 {
3     char *str = vmalloc(1000 * sizeof(char));
4     int cnt;
```

## Продолжение листинга 4.8

```

5
6     int count = 0;
7     struct Node* head = kmalloc(sizeof(Node), GFP_KERNEL);
8     head->task = NULL;
9     head->next = NULL;
10    struct Node* cur = head;
11    struct list_head* pos;
12
13    list_for_each(pos, &task->children)
14    {
15        if (head->task == NULL)
16            head->task = list_entry(pos, struct task_struct, sibling);
17        else
18        {
19            cur->next = kmalloc(sizeof(Node), GFP_KERNEL);
20            cur->next->task = list_entry(pos, struct task_struct, sibling);
21            cur->next->next = NULL;
22            cur = cur->next;
23        }
24
25        count++;
26    }
27
28    printk("process: %s[%d] (parent: %s[%d])\n", task->comm, task->pid ,
↪ task->parent->comm, task->parent->pid);
29    cnt = sprintf(str, "process: %s[%d] (parent: %s[%d])\n",
30                  task->comm, task->pid , task->parent->comm, task->parent->pid);
31
32    str[cnt] = '\0';
33    memcpy(buf + (*offset), str, strlen(str));
34    (*offset) += strlen(str);
35
36    if(count > 0)
37    {
38        struct Node* pr;
39        n = n - 1;
40        int i = 1;
41
42        for(pr = head; pr != NULL;)
43        {
44            int m = DEPTH;
45
46            for(; m > n; m--)
47            {
48                printk("\t");
49                memcpy(buf + (*offset), "\t", strlen("\t"));
50                (*offset) += strlen("\t");

```

## Продолжение листинга 4.8

```
51         }
52
53         printk("--- CHILD: %d, ", i);
54         cnt = sprintf(str, "--- CHILD: %d, ", i);
55
56         str[cnt] = '\0';
57         memcpy(buf + (*offset), str, strlen(str));
58         (*offset) += strlen(str);
59
60         process_info(pr->task, n, buf, offset);
61
62         i = i+1;
63         struct Node* temp = pr;
64         pr = pr->next;
65
66         kfree(temp);
67         temp = NULL;
68     }
69 }
70
71     buf[*offset] = '\0';
72     vfree(str);
73 }
```