Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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:*

## *«Разработка модуля ядра, реализующего хранилище типа «ключ-значение»»*

Студент ИУ7-71Б (Группа) _____ (Подпись, дата) Гаврилюк В. А. (И. О. Фамилия)

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

*2026 г.*

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

---

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
_____ И. В. Рудаков
«__» сентября 2025 г.

# ЗАДАНИЕ

## на выполнение курсовой работы

по дисциплине Операционные системы

тема курсовой работы

**Разработка модуля ядра, реализующего хранилище типа «ключ-значение»**

Студент группы **ИУ7-71Б**

**Гаврилюк Владислав Анатольевич**

График выполнения КР: 25% к 5 нед., 50% к 8 нед., 75% к 11 нед., 100% к 15 нед.

***Техническое задание***

*Разработать модуль ядра, реализующий хранилище типа «ключ-значение» для обмена данными между процессами режима пользователя через символьное виртуальное устройство. Необходимо предоставить процессам интерфейс для записи и чтения данных.*

***Оформление курсовой работы:***

Расчетно-пояснительная записка на 12-32 листах формата А4.

***Перечень графического (иллюстративного) материала:***

Презентация на 8-16 слайдах.

Дата выдачи задания «__» сентября 2025 г.

**Руководитель курсовой работы** _____ **Н. Ю. Рязанова**

**Студент** _____ **В. А. Гаврилюк**

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

При разработке программного обеспечения для операционной системы Linux возникает необходимость хранения временных служебных данных, используемых несколькими процессами. Такие данные, как правило, идентифицируются по ключу, имеют ограниченное время актуальности и могут быть восстановлены или пересчитаны при необходимости. Критическим требованием в подобных сценариях является устойчивость системы при росте нагрузки и контролируемое использование ресурсов.

В Linux для этой цели может быть использована разделяемая память, однако она не решает ряд практических задач, возникающих при совместном использовании данных. Синхронизация доступа к разделяемой памяти требует использования дополнительных средств взаимоисключения, которые должны быть предварительно согласованы между независимыми процессами. Размер сегмента разделяемой памяти является фиксированным и задаётся при создании, а управление заполнением и поведением при достижении предельного объема данных возлагается на пользовательские приложения, что усложняет их реализацию.

Цель курсовой работы — разработка модуля ядра, реализующего хранилище типа «ключ-значение» для обмена данными между процессами режима пользователя через символьное виртуальное устройство.

# 1 Аналитический раздел

## 1.1 Постановка задачи

В соответствии с заданием на курсовую работу требуется разработать модуль ядра, реализующий хранилище типа «ключ-значение» для обмена данными между процессами режима пользователя через символьное виртуальное устройство.

Для выполнения работы требуется решить следующие задачи:

1. Проанализировать известные способы хранения пар «ключ-значение», выбрать наиболее подходящий для реализации хранилища;
2. Проанализировать известные стратегии вытеснения данных;
3. Разработать алгоритмы и структуру ПО;
4. Разработать программное обеспечение;
5. Провести исследование разработанного ПО.

## 1.2 Способы хранения пар «ключ-значение»

Наиболее распространенными структурами данных для хранения пар «ключ-значение» являются:

– хеш-таблицы;

– сбалансированные деревья;

– таблицы с прямой адресацией.

### 1.2.1 Таблицы с прямой адресацией

Таблицы с прямой адресацией представляют собой массивы, в которых каждый индекс напрямую соответствует ключу. Асимптотическая сложность операций вставки, поиска, удаления в худшем случае — $O(1)$, однако данный метод применим только при ограниченном диапазоне числовых ключей и становится непрактичным при произвольных строковых или бинарных ключах,

поскольку потребление памяти напрямую зависит от размера ключевого пространства [1].

Прямая адресация редко используется в полноценных хранилищах типа «ключ-значение» и в основном ограничена специализированными задачами с небольшим числом ключей.

### 1.2.2 Хеш-таблицы

Альтернативой таблиц с прямой адресацией являются хеш-таблицы, в которых ключи преобразуются с помощью хеш-функций, что позволяет использовать произвольные ключи. Средняя асимптотическая сложность операций поиска, вставки и удаления составляет $O(1)$ [1], то есть время выполнения операций не зависит от количества элементов в таблице.

Основным недостатком хеш таблиц является возможность неравномерного распределения ключей и возникновения коллизий при плохо подобранной хеш-функции.

#### Открытая адресация

При открытой адресации все элементы хранятся непосредственно в массиве хеш-таблицы. В случае коллизии выполняется поиск альтернативной позиции в массиве в соответствии с выбранным алгоритмом пробирования. К данному классу относятся реализации с линейным и квадратичным пробированием, двойным хешированием и прочие.

Корректная работа хеш-таблиц с открытой адресацией требует ограничения коэффициента заполнения массива, поскольку при его росте резко увеличивается среднее время поиска и вставки. Удаление элементов усложняется необходимостью использования специальных маркеров, предотвращающих нарушение последовательности пробирования.

#### Закрытая адресация

При закрытой адресации каждая ячейка массива хеш-таблицы содержит указатель на внешнюю структуру данных, в которой хранятся все элементы, соответствующие одному значению хеш-функции. На практике в качестве такой структуры чаще всего используется связный список, а ячейка хеш-таблицы содержит структуру с указателем на голову списка.

Данный подход допускает хранение произвольного количества элементов независимо от размера массива хеш-таблицы. Корректность работы хеш-таблицы с закрытой адресацией не зависит от коэффициента заполнения массива корзин, однако при неравномерном распределении ключей возможен рост длины отдельных списков, что приводит к увеличению времени поиска.

В заголовочном файле `linux/hashtable.h` [2] определен набор структур и макросов, предоставляющий универсальный интерфейс для реализации хеш-таблицы с закрытой адресацией. Каждая корзина хеш-таблицы представлена в виде структуры `struct hlist_head`, указанной на листинге 1.1. Она содержит указатель на первый элемент связного списка.

Листинг 1.1 – Структура struct hlist_head

```
struct hlist_head {
    struct hlist_node *first;
};
```

Узел связного списка определен структурой `struct hlist_node`, представленной на листинге 1.2. Каждый узел соответствует только одному элементу в корзине.

Листинг 1.2 – Структура struct hlist_node

```
struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

### 1.2.3 Сбалансированные деревья

Сбалансированные деревья представляют собой иерархические структуры данных, в которых высота поддеревьев каждого узла контролируется для обеспечения гарантированной вычислительной сложности операций.

Сбалансированные деревья применяются в случаях, когда требуется поддержка упорядоченного доступа к ключам или диапазонных запросов. Вставка, поиск и удаление в таких структурах имеют асимптотическую временную сложность $O(\log n)$, которая не зависит от распределения ключей.

### 1.2.4 Выбор способа хранения пар «ключ-значение»

В таблице 1.1 приведен сравнительный анализ рассмотренных способов хранения пар «ключ-значение».

Таблица 1.1 – Сравнение основных методов реализации «in-memory» хранилищ «ключ-значение»

| Критерий сравнения | Прямая адресация | Хеш-таблицы | Сбалансированные деревья |
|---|---|---|---|
| Асимптотическая сложность операций | $O(1)$ | $O(1)$ | $O(\log n)$ |
| Поддержка произвольных ключей | – | + | + |
| Поддержка диапазонных запросов | – | – | + |

В результате сравнительного анализа структур данных для реализации хранилища типа «ключ-значение» были выбраны хеш-таблицы, так как они поддерживают произвольные ключи и имеют асимтотическую слоность операций $O(1)$, а диапазонные запросы в рамках технического задания на курсовую работу не требуются.

## 1.3 Стратегии вытеснения данных

### 1.3.1 Стратегия FIFO

Стратегия FIFO (First In, First Out) предполагает удаление элементов в порядке их добавления в хранилище. Алгоритм работы данной стратегии основан на поддержании очереди, в которой каждый новый элемент помещается в конец структуры, а при необходимости вытеснения удаляется элемент из её начала. FIFO не учитывает ни частоту, ни давность использования данных после их добавления, что может приводить к удалению активно используемых элементов.

Основным преимуществом данной стратегии является простота реализации и минимальные накладные расходы, однако низкая адаптивность к реальному характеру нагрузки ограничивает её применимость в системах с неравномерным распределением обращений.

### 1.3.2 Стратегия LFU

Стратегия LFU (Least Frequently Used) ориентирована на удаление элементов с наименьшим числом обращений. Для каждого элемента хранилища поддерживается счётчик, увеличиваемый при каждом обращении. При вытеснении выбирается элемент с минимальным значением счётчика. Существенным недостатком LFU является слабая адаптация к изменению характера нагрузки во времени, так как элементы, активно используемые на ранних этапах работы, могут сохранять высокие значения счётчиков и вытеснять более актуальные данные.

### 1.3.3 Стратегия LRU

Стратегия LRU (Least Recently Used) основывается на времени последнего обращения к элементам и предполагает удаление данных, которые не использовались на протяжении наибольшего временного интервала. Реализации данной стратегии обычно основаны двусвязном списке, и все основные операции — поиск, обновление позиции и удаление — могут быть реализованы с асимптотической сложностью $O(1)$.

### 1.3.4 Сравнение стратегий вытеснения

В таблице 1.2 приведен сравнительный анализ рассмотренных стратегий.

Таблица 1.2 – Сравнение стратегий вытеснения данных

| Критерий | FIFO | LFU | LRU |
|---|---|---|---|
| Критерий вытеснения | Порядок добавления | Частота обращений | Время последнего доступа |
| Риск удержания устаревших элементов | – | + | – |
| Риск вытеснения недавно используемых элементов | + | + | – |

Выбор стратегии вытеснения в основном определяется требованиями конкретной системы. Однако в качестве универсальной была выбрана страте-

гия LRU в силу отсутствия удержания устаревших данных и вытеснения более актуальных. Также стоит отметить широкую распространенность данной стратегии в ядре Linux.

## 1.4 Символьное устройство

Символьные устройства предоставляют стандартный интерфейс для взаимодействия с модулем ядра через файловые операции. Основной структурой для регистрации символьного устройства является `struct cdev` [3]:

Листинг 1.3 – Структура struct cdev

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

Однако ядро Linux также предоставляет упрощенный интерфейс `struct miscdevice` [4], продемонстрированный на листинге 1.4. В этом случае старший номер уже зарезервирован ядром для всех misc-устройств, и каждому устройству назначается уникальный младший номер. Регистрация выполняется через функцию `misc_register`.

Листинг 1.4 – Структура struct miscdevice

```
struct miscdevice  {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};
```

Набор файловых операций описывается структурой `struct file_operations` [5]:

Листинг 1.5 – Структура file_operations

```
struct file_operations {
    struct module *owner;
    // ...
    ssize_t (*read) (struct file *, char __user *, size_t,
       loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
       size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
       unsigned long);
    // ...
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    // ...
};
```

Системный вызов ioctl [6] предназначен для передачи управляющих команд, каждая из которых имеет явный числовой идентификатор и связанную с ним структуру данных. Такой подход позволяет формально разделять операции вставки, поиска и удаления элементов хранилища и обеспечивает строгую типизацию передаваемых данных.

## Вывод

В результате проведённого анализа для хранения пар «ключ-значение» были выбраны хеш-таблицы с закрытой адресацией, так как они поддерживают произвольные ключи и обеспечивают амортизированную асимптотическую сложность операций доступа $O(1)$.

На основе анализа стратегий вытеснения данных в качестве основной была выбрана стратегия LRU в силу асимптотической сложности операций доступа $O(1)$ и отсутствия удержания устаревших элементов и вытеснения недавно используемых.

Для регистрации символьного виртуального устройства был выбран упрощенный интерфейс `struct miscdevice`, так как он не требует регистрации старшего номера устройства.

# 2 Конструкторский раздел

## 2.1 Функциональная модель системы

На рисунке 2.1 представлена IDEF0-диаграмма нулевого уровня разрабатываемого модуля ядра.

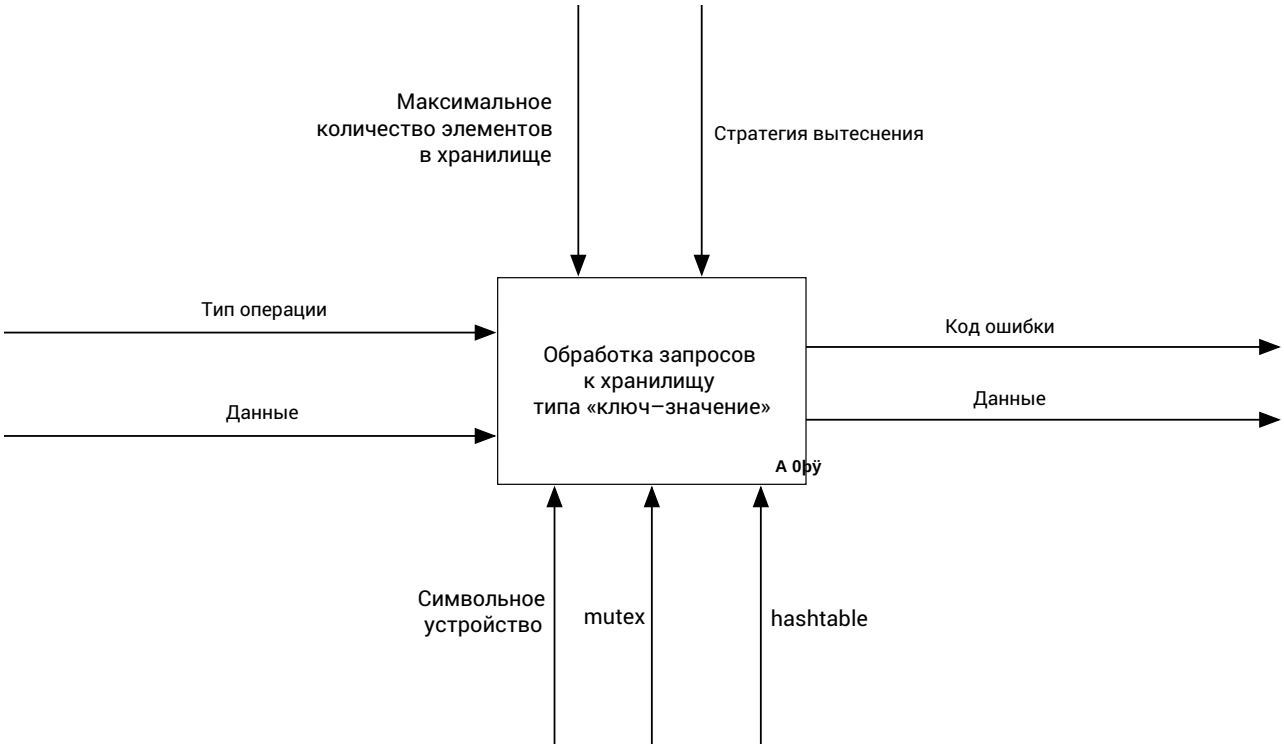


Рисунок 2.1 – Функциональная модель системы в нотации IDEF0 (A0)

На рисунке 2.2 представлена IDEF0-диаграмма первого уровня разрабатываемого модуля ядра.

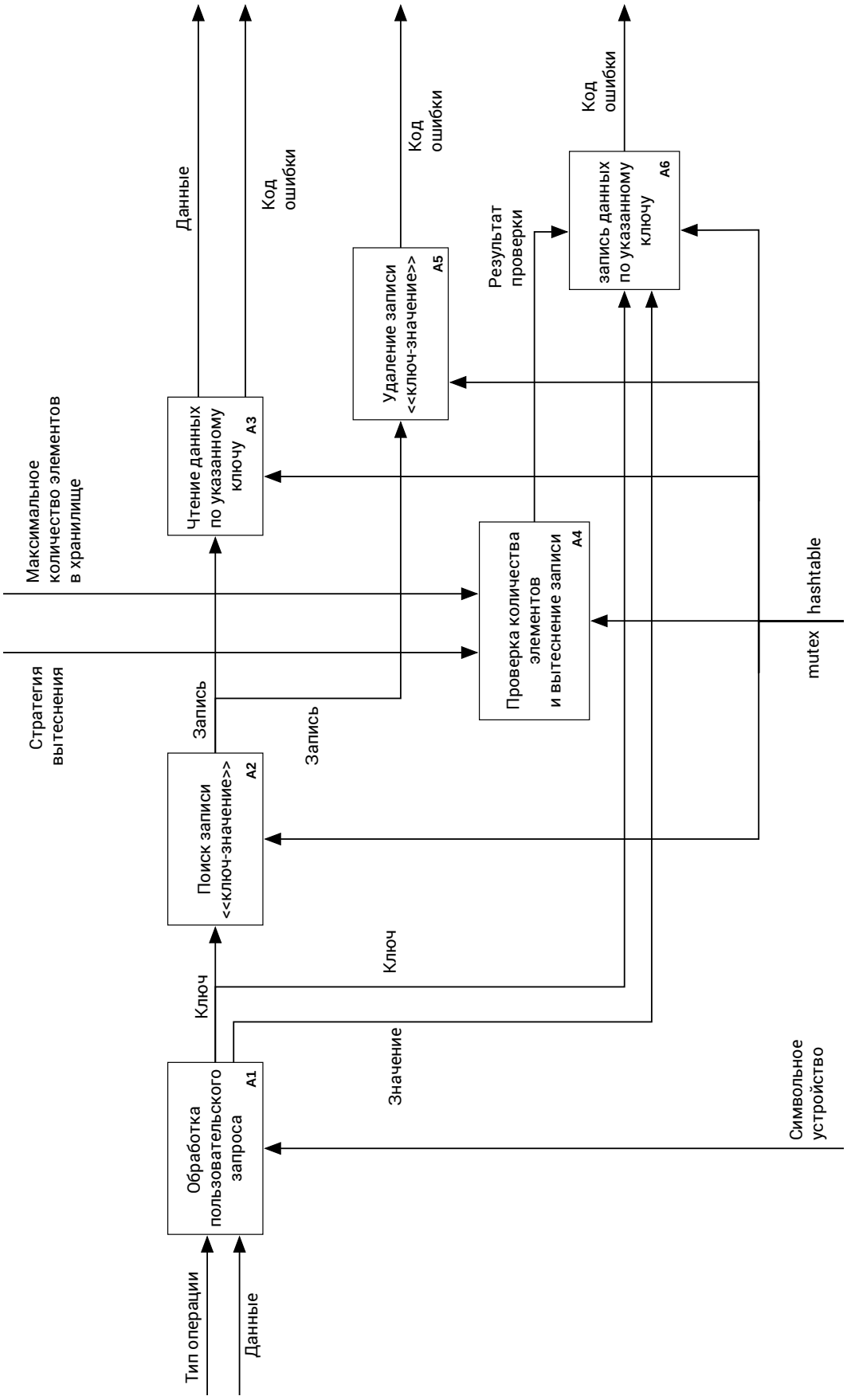


Рисунок 2.2 – Функциональная модель системы в нотации IDEF0 (A1)

## 2.2 Схемы алгоритмов

На рисунке 2.3 представлена схема алгоритма записи значения по ключу.

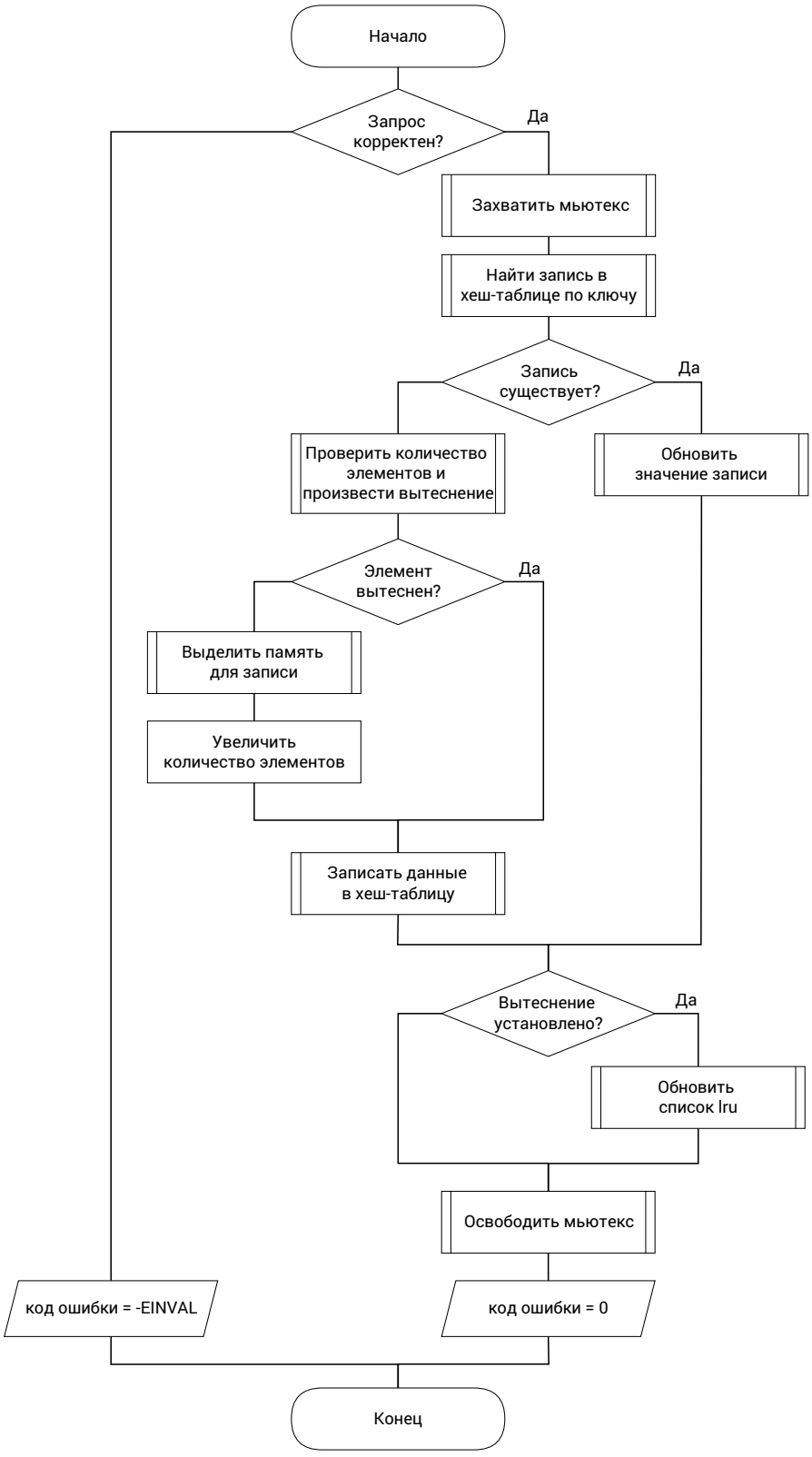


Рисунок 2.3 – Схема алгоритма записи значения по ключу

На рисунке 2.4 приведена схема алгоритма чтения значения.

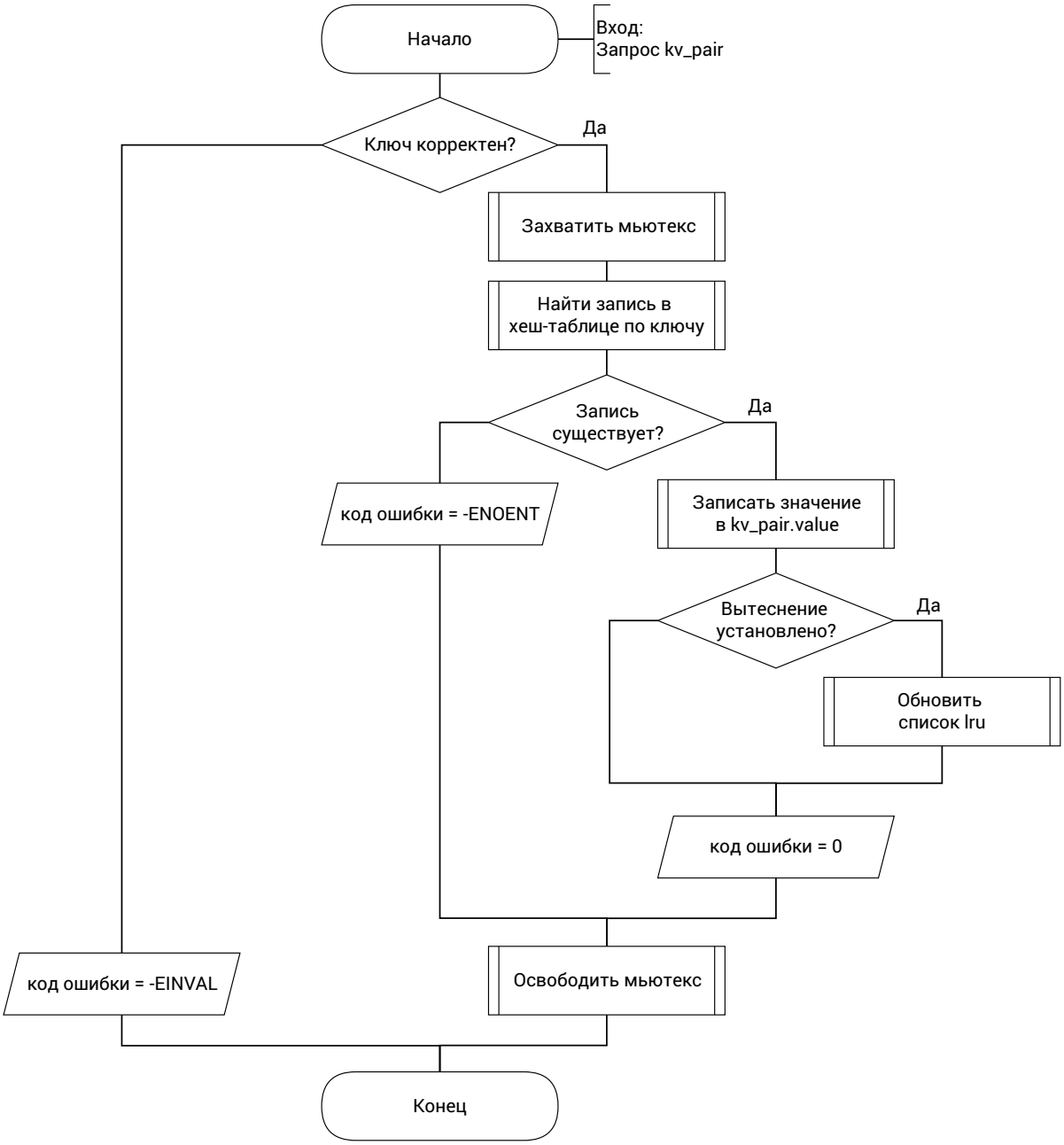


Рисунок 2.4 – Схема алгоритма чтения значения

На рисунке 2.5 приведена схема алгоритма удаления значения.

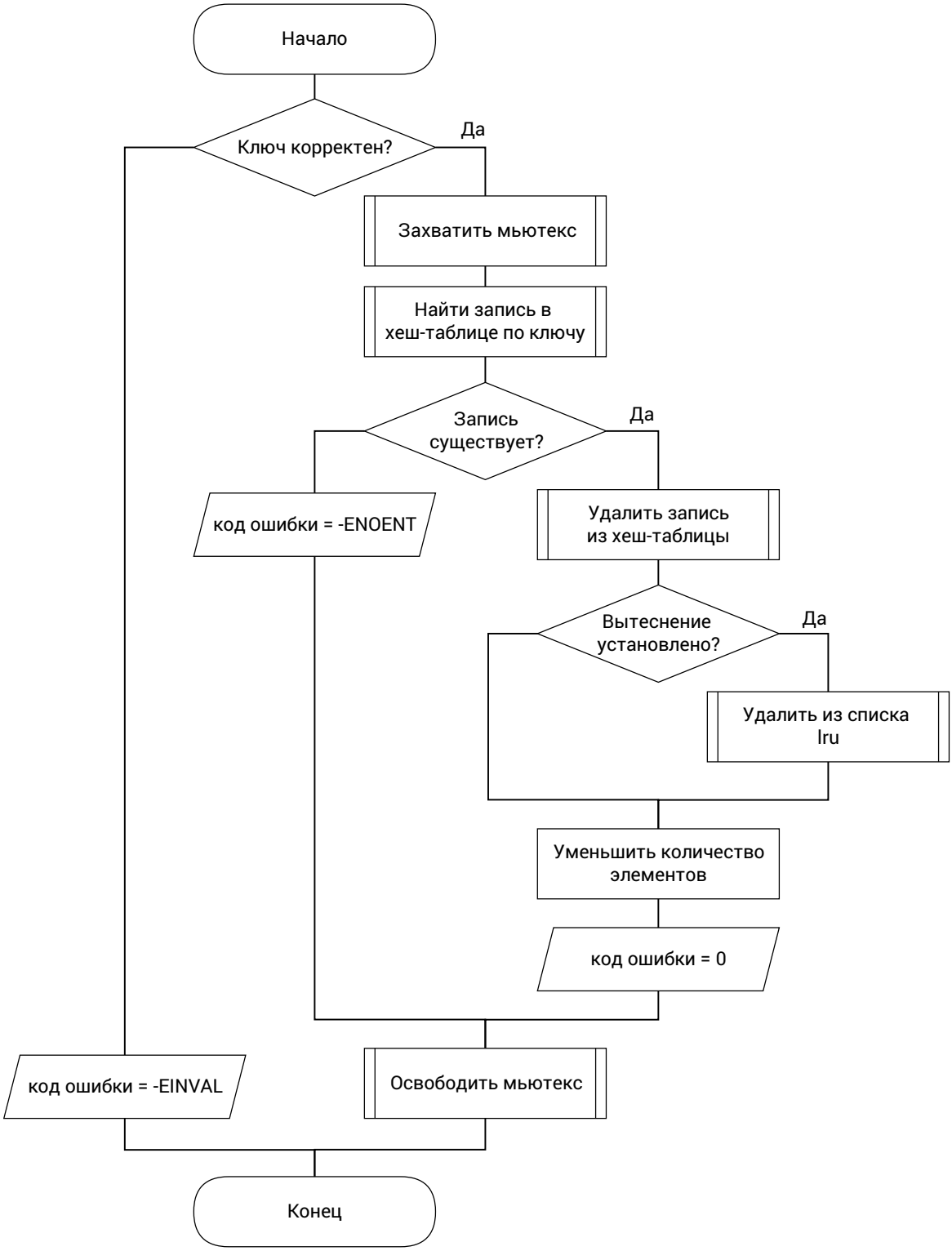


Рисунок 2.5 – Схема алгоритма удаления значения

## 2.3 Структура программного обеспечения

На рисунке 2.6 представлена структура програмного обеспечения.

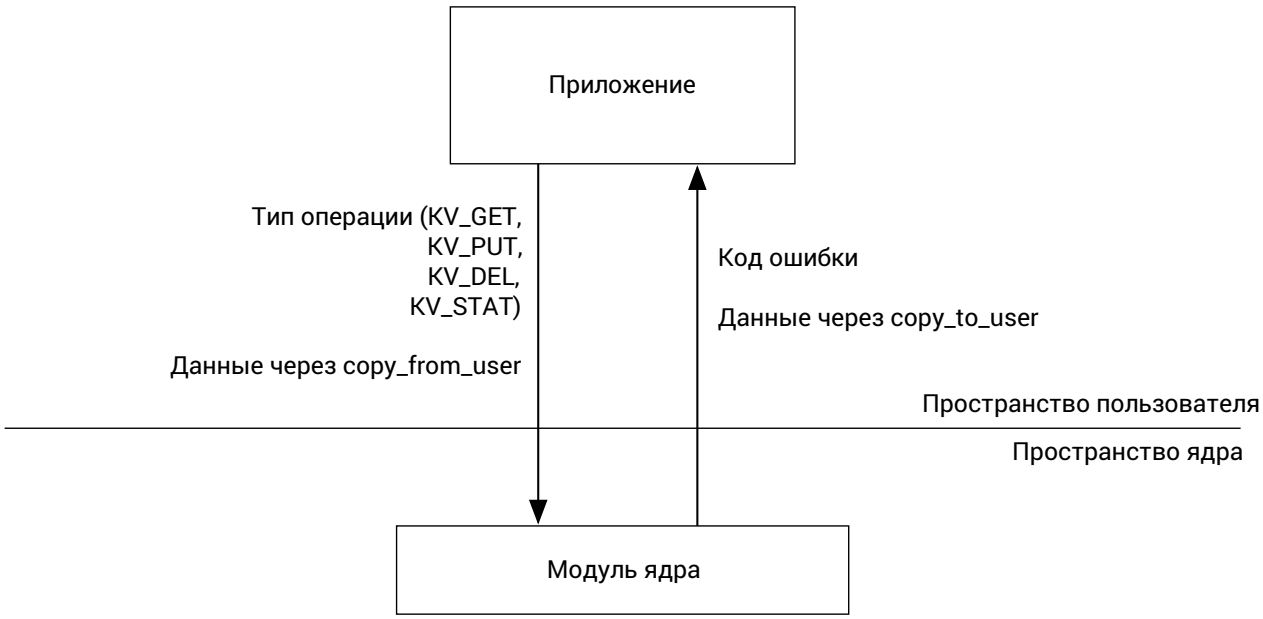


Рисунок 2.6 – Структура програмного обеспечения.

# 3 Технологический раздел

## 3.1 Выбор языка и среды программирования

Для реализации загружаемого модуля был выбран язык программирования (ЯП) C, так как он является основным языком для разработки ядра Linux. Также для разработки модулей постепенно внедряется язык Rust, однако его использование ограничено из-за неполной поддержки низкоуровневых возможностей ядра и нестабильности API. В качестве системы сборки была выбрана утилита make.

## 3.2 Реализация структур данных

На листинге 3.1 представлена реализация структур для передачи ключей и значений.

Листинг 3.1 – Структуры для предачи ключей и значений

```
#define KV_MAX_KEY   64
#define KV_MAX_VAL   256

struct kv_key {
    char data[KV_MAX_KEY];
    kv_u32 len;
};

struct kv_value {
    char data[KV_MAX_VAL];
    kv_u32 len;
};

struct kv_pair {
    struct kv_key key;
    struct kv_value value;
};
```

На листинге 3.2 представлены основные структуры хранилища типа «ключ-значение».

Листинг 3.2 – Основные структуры хранилища

```
struct kv_item {
    struct hlist_node hnode;
    struct list_head lru_node;
    struct kv_key key;
    struct kv_value value;
};

struct kv_bucket {
    struct hlist_head head;
};

struct kv_lru {
    struct list_head head;
};

struct kv_store {
    struct kv_bucket *buckets;
    struct kv_lru lru;
    struct mutex lock;

    kv_u8 use_lru;
    kv_u64 bucket_count;
    kv_u64 max_items;
    atomic_t curr_items;
};
```

На листинге 3.3 представлена реализация структуры для передачи информации о хранилище: количество корзин в хранилище, максимальное количество элементов, количество элементов в хранилище на момент запроса и включено ли вытеснение элементов.

Листинг 3.3 – Структура `struct kv_stat`

```
struct kv_usage_stat {
    kv_u64 bucket_count;
    kv_u64 max_items;
    kv_u64 curr_items;
    kv_u8  use_lru;
};
```

## 3.3 Реализация функций хранилища

На листингах 3.4 — 3.6 представлена реализация функций записи, чтения и удаления значений.

Листинг 3.4 – Функция записи значения по ключу

```
int kv_put(struct kv_store *s, struct kv_pair *p)
{
    if (s == NULL || p == NULL || p->key.len > KV_MAX_KEY
        || p->value.len > KV_MAX_VAL)
        return -EINVAL;
    u32 hash = kv_key_hash(&p->key) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];
    struct kv_item *item;

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, &p->key);
    if (item) {
        kv_item_set_value(item, p);
        if (s->use_lru)
            lru_touch(&s->lru, item);
        mutex_unlock(&s->lock);
        return 0;
    }
    item = kv_check_and_evict_item(s);
    if (!item) {
        item = kmalloc(sizeof(*item), GFP_KERNEL);
        if (!item) {
            mutex_unlock(&s->lock);
            return -ENOMEM;
        }
        atomic_inc(&s->curr_items);
    }

    kv_item_init(item, p);
    hlist_add_head(&item->hnode, &b->head);
    if (s->use_lru) {
        INIT_LIST_HEAD(&item->lru_node);
        lru_touch(&s->lru, item);
    }
    mutex_unlock(&s->lock);
```

```
    return 0;
}
```

Листинг 3.5 – Функция чтения значения по ключу

```
int kv_get(struct kv_store *s, struct kv_pair *p)
{
    if (s == NULL || p == NULL || p->key.len > KV_MAX_KEY)
        return -EINVAL;

    int err = 0;
    u32 hash = kv_key_hash(&p->key) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];
    struct kv_item *item;

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, &p->key);
    if (item) {
        kv_pair_set_value(p, item);
        if (s->use_lru)
            lru_touch(&s->lru, item);
    } else {
       err = -ENOENT;
    }
    mutex_unlock(&s->lock);

    return err;
}
```

Листинг 3.6 – Функция удаления значения

```
int kv_del(struct kv_store *s, struct kv_key *k)
{
    if (s == NULL || k == NULL || k->len > KV_MAX_KEY)
        return -EINVAL;

    int err = 0;
    struct kv_item *item;
    u32 hash = kv_key_hash(k) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, k);
    if (item) {
```

```
        hlist_del(&item->hnode);
        if (s->use_lru)
            lru_remove(&s->lru, item);
        kfree(item);
        atomic_dec(&s->curr_items);
    } else {
        err = -ENOENT;
    }
    mutex_unlock(&s->lock);

    return err;
}
```

## 3.4 Реализация LRU

На листинге 3.7 продемонстрирована функция для перемещения элемента в начало списка, а также функции удаления и вытеснения.

Листинг 3.7 – Функции для работы с LRU списком

```
void lru_touch(struct kv_lru *lru, struct kv_item *item)
{
    if (list_empty(&item->lru_node))
        list_add(&item->lru_node, &lru->head);
    else
        list_move(&item->lru_node, &lru->head);
}

void lru_remove(struct kv_lru *lru, struct kv_item *item)
{
    if (!list_empty(&item->lru_node))
        list_del_init(&item->lru_node);
}

struct kv_item *lru_evict(struct kv_lru *lru)
{
    struct kv_item *victim = NULL;

    if (!list_empty(&lru->head)) {
        victim = list_last_entry(&lru->head, struct kv_item,
           lru_node);
        list_del_init(&victim->lru_node);
    }
```

```
    return victim;
}
```

## 3.5 Обработчик ioctl

На листинге 3.8 представлена функция обработки операций `ioctl`, которая затем передается в структуру `struct file_operations`.

Листинг 3.8 – Функция обработки операций ioctl

```
static long kv_ioctl(struct file *f, unsigned int cmd, unsigned
   long arg)
{
    int err;
    struct kv_pair p;
    struct kv_key k;
    struct kv_usage_stat stat;

    switch (cmd) {
    case KV_PUT:
        if (copy_from_user(&p, (void __user *)arg, sizeof(p)))
            return -EFAULT;
        pr_info("kv_ioctl: KV_PUT key=%.*s value=%.*s\n",
            (int) p.key.len, p.key.data, (int) p.value.len,
               p.value.data);

        return kv_put(&store, &p);

    case KV_GET:
        if (copy_from_user(&p, (void __user *)arg, sizeof(p)))
            return -EFAULT;

        pr_info("kv_ioctl: KV_GET key=%.*s\n", (int) p.key.len,
           p.key.data);
        err = kv_get(&store, &p);
        if (err != 0)
            return err;

        return copy_to_user((void __user *)arg, &p, sizeof(p));

    case KV_DEL:
        if (copy_from_user(&k, (void __user *)arg, sizeof(k)))
```

```
            return -EFAULT;
        pr_info("kv_ioctl: KV_DEL key=%.*s\n", (int) k.len,
           k.data);

        return kv_del(&store, &k);

    case KV_STAT:
        pr_info("kv_ioctl: KV_STAT");
        err = kv_stat(&store, &stat);
        if (err != 0)
            return err;

        return copy_to_user((void __user *)arg, &stat,
           sizeof(stat));

    default:
        pr_info("kv_ioctl: unknown command %u\n", cmd);
        return -EINVAL;
    }
}
```

Листинг 3.9 – Инициализация структуры `struct file_operations`

```
const struct file_operations kv_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = kv_ioctl,
};
```

## 3.6 Регистрация символьного устройства

На листинге 3.10 представлена инициализированная структура `struct miscdevice`. Регистрация символьного устройства выполняется при инициализации модуля:

Листинг 3.10 – Регистрация символьного устройства

```
static struct miscdevice kv_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "kvstore",
    .fops = &kv_fops,
};

static int __init kv_init(void)
{
```

```
    pr_info("kv_kernel: init storage");
    int err = kv_store_init(&store, buckets, max_items, use_lru);
    if (err) {
        pr_err("kv_kernel: init failed");
        return err;
    }
    err = misc_register(&kv_dev);
    if (err) {
        pr_err("kv_kernel: misc_register failed");
        return err;
    }
    pr_info("kv_kernel: init storage OK");

    return 0;
}
```

## 3.7 Конфигурация параметров хранилища

Конфигурирование хранилища реализовано с помощью параметров загрузки модуля. Параметры определяются макросом `module_param(name, type, perm)`, продемонстрированного на листинге 3.11.

Листинг 3.11 – Конфигурация параметров хранилища

```
#define BUCKETS_DEFAULT 64
#define MAX_ITEMS_DEFAULT 1024
#define USE_LRU_DEFAULT 0

static int buckets = BUCKETS_DEFAULT;
static int max_items = MAX_ITEMS_DEFAULT;
static bool use_lru = USE_LRU_DEFAULT;

module_param(buckets, int, 0444);
module_param(max_items, int, 0444);
module_param(use_lru, bool, 0444);
```

Полный исходный код реализованного модуля ядра находится в приложении А.

# 4 Исследовательский раздел

Исследование работоспособности разработанного модуля ядра проводилось на ядре Linux версии 6.14, дистрибутив — Ubuntu 24.04.3 LTS.

Разработано два тестовых приложения для проверки всех основных функций хранилища при однопоточном и многопоточном доступе. Исходный код тестовых программ приведен в приложении В. Для взаимодействия с хранилищем, программы используют разработанную пользовательскую библиотеку, исходный код которой представлен в приложении Б.

## 4.1 Однопоточный доступ

Конфигурация хранилища для данного исследования:

- buckets=5 — количество корзин;
- max_items=10 — максимальное количество элементов в хранилище;
- use_lru=1 — режим работы с вытеснением (LRU).

Тестовое приложение выполняет следующую последовательность операций:

1. запрос информации о хранилище;
2. сохранение 11 элементов с уникальными ключами;
3. запрос информации о хранилище;
4. проверка отсутствия первого записанного элемента;
5. получение значения второго записанного элемента;
6. удаление второго записанного элемента;
7. проверка отсутствия второго записанного элемента;
8. запрос информации о хранилище.

Результат работы тестового приложения представлен на рисунке 4.1.

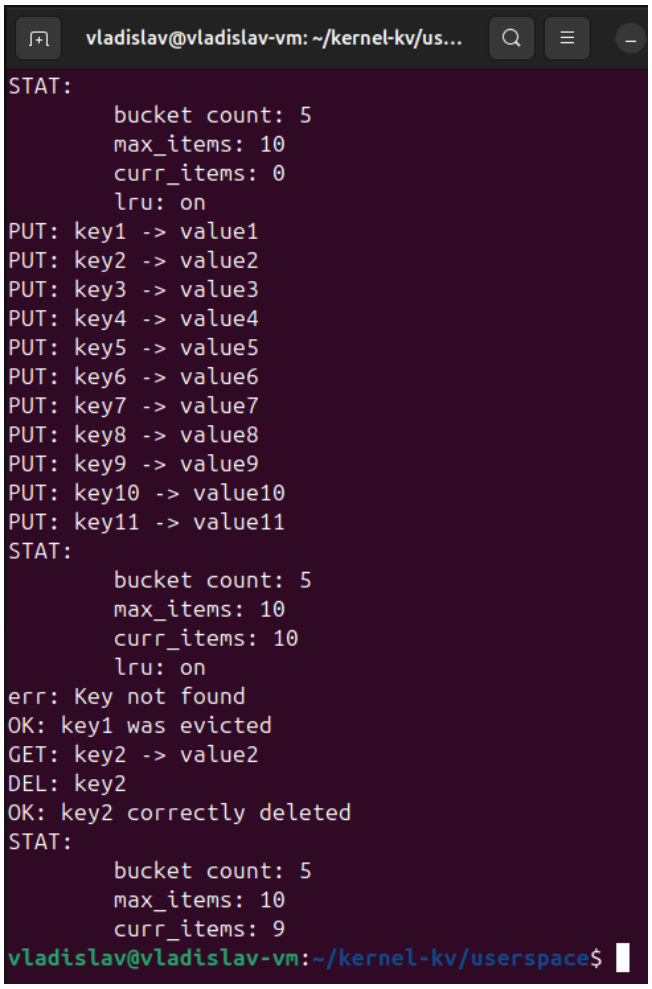


Рисунок 4.1 – Результат работы тестового приложения №1

На основе полученных результатов можно сделать вывод, что хранилище корректно обрабатывает ситуацию достижения максимального количества элементов и реализует вытеснение согласно стратегии LRU. Все операции записи, чтения и удаления соответствует заданным требованиям.

## 4.2 Многопоточный доступ

Для данного исследования используется стандартная конфигурация хранилища:

- buckets=64 — количество корзин;
- max_items=1024 — максимальное количество элементов в хранилище;
- use_lru=0 — режим работы без вытеснения.

Тестовое приложение в одном потоке циклически обновляет значения для фиксированного набора ключей, и в четырех потоках выполняется чтение по данным ключам. Результат работы тестового приложения представлен на рисунке 4.2.

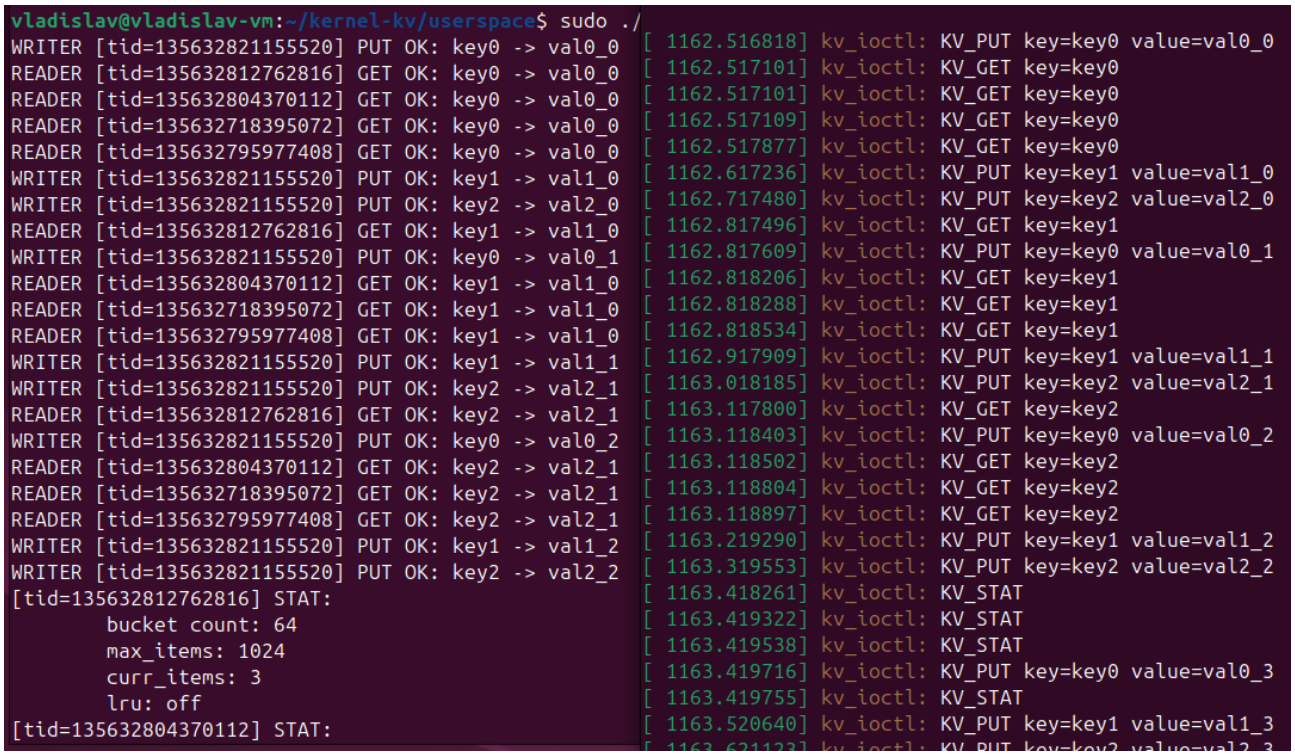


Рисунок 4.2 – Результат работы тестового приложения №2

Исследование показало, что хранилище корректно обрабатывает операции чтения и записи в условиях конкруретного доступа из нескольких потоков: все потоки-читатели получают актуальные значения.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были проанализированы способы организации хранилищ типа «ключ–значение», стратегии вытеснения данных, способы регистрации символьного устройства. На основе проведенного анализа для реализации хранилища были выбраны хеш-таблицы, стратегия вытеснения LRU и интерфейс `miscdevice` для регистрации символьных устройств.

Были разработаны алгоритмы чтения, записи и удаления данных. Разработанный модуль ядра поддерживает стратегию вытеснения LRU, что позволит упростить реализацию приложений за счет устранения необходимости самостоятельного управления памятью.

Исследование разработанного модуля показало, что он полностью отвечает техническому заданию и корректно решает все поставленные задачи.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Алгоритмы: построение и анализ / Т. Х. Кормен [и др.] ; пер. П. с англ. — 2-е издание. — Москва : Издательский дом “Вильямс”, 2011. — С. 1296. — ISBN 978-5-8459-0857-. — Парал. тит. англ.
2. Linux kernel source code: hashtable.h [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/hashtable.h` (дата обращения: 29.11.2025).
3. Linux kernel source code: struct cdev [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/cdev.h#L15` (дата обращения: 29.11.2025).
4. Linux kernel source code: struct miscdevice [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/miscdevice.h#L53` (дата обращения: 29.11.2025).
5. Linux kernel source code: struct file_operations [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/fs.h#L2132` (дата обращения: 29.11.2025).
6. Linux kernel source code: ioctl system call interface [Электронный ресурс]. — Режим доступа: `https://elixir.bootlin.com/linux/v6.18.6/source/include/linux/fs.h#L2200` (дата обращения: 29.11.2025).

# ПРИЛОЖЕНИЕ А
## Исходный код модуля ядра

Листинг А.1 – Исходный код файла kv_types.h

```
#ifndef KV_TYPES_H
#define KV_TYPES_H

#include <linux/types.h>
#include <linux/mutex.h>

#define KV_MAX_KEY   64
#define KV_MAX_VAL   256

typedef __u8 kv_u8;
typedef __u32 kv_u32;
typedef __u64 kv_u64;

struct kv_key {
    char data[KV_MAX_KEY];
    kv_u32 len;
};

struct kv_value {
    char data[KV_MAX_VAL];
    kv_u32 len;
};

struct kv_pair {
    struct kv_key key;
    struct kv_value value;
};

struct kv_usage_stat {
    kv_u64 bucket_count;
    kv_u64 max_items;
    kv_u64 curr_items;
    kv_u8  use_lru;
};

struct kv_item {
    struct hlist_node hnode;
```

```
    struct list_head lru_node;
    struct kv_key key;
    struct kv_value value;
};

struct kv_bucket {
    struct hlist_head head;
};

struct kv_lru {
    struct list_head head;
};

struct kv_store {
    struct kv_bucket *buckets;
    struct kv_lru lru;
    struct mutex lock;

    kv_u8 use_lru;
    kv_u64 bucket_count;
    kv_u64 max_items;
    atomic_t curr_items;
};

#endif // KV_TYPES_H
```

Листинг А.2 – Исходный код файла kv_ioctl.h

```
#ifndef KV_IOCTL_H
#define KV_IOCTL_H

#include <linux/ioctl.h>
#include "kv_types.h"

#define KV_IOC_MAGIC 'k'

#define KV_PUT _IOW(KV_IOC_MAGIC, 1, struct kv_pair)
#define KV_GET _IOWR(KV_IOC_MAGIC, 2, struct kv_pair)
#define KV_DEL _IOW(KV_IOC_MAGIC, 3, struct kv_key)
#define KV_STAT _IOR(KV_IOC_MAGIC, 4, struct kv_usage_stat)

#endif // KV_IOCTL_H
```

Листинг А.3 – Исходный код файла kv_store.h

```
#ifndef KV_STORE_H
#define KV_STORE_H

#include <linux/hashtable.h>
#include <linux/mutex.h>
#include <linux/jhash.h>
#include <linux/string.h>
#include <linux/slab.h>
#include "kv_types.h"
#include "kv_lru.h"

#define BUCKETS_DEFAULT 64
#define MAX_ITEMS_DEFAULT 1024
#define USE_LRU_DEFAULT 0

int kv_store_init(struct kv_store *s, size_t buckets, size_t
   max_items, bool lru);
void kv_store_destroy(struct kv_store *s);

int kv_put(struct kv_store *s, struct kv_pair *p);
int kv_get(struct kv_store *s, struct kv_pair *p);
int kv_del(struct kv_store *s, struct kv_key *k);
int kv_stat(struct kv_store *store, struct kv_usage_stat *stat);

#endif // KV_STORE_H
```

Листинг А.4 – Исходный код файла kv_store.c

```
#include "kv_store.h"

inline static u32 kv_key_hash(struct kv_key *key)
{
    return jhash(key->data, key->len, 0);
}

inline static int kv_key_equal(struct kv_key *key1, struct
   kv_key *key2)
{
    return key1->len == key2->len && memcmp(key1->data,
       key2->data, key1->len) == 0;
}
```

```
inline static void kv_item_init(struct kv_item *item, struct
   kv_pair *p)
{
    memcpy(item->key.data, p->key.data, p->key.len);
    memcpy(item->value.data, p->value.data, p->value.len);
    item->key.len = p->key.len;
    item->value.len = p->value.len;
}

inline static void kv_item_set_value(struct kv_item *item,
   struct kv_pair *p)
{
    memcpy(item->value.data, p->value.data, p->value.len);
    item->value.len = p->value.len;
}

inline static void kv_pair_set_value(struct kv_pair *p, struct
   kv_item *item)
{
    memcpy(p->value.data, item->value.data, item->value.len);
    p->value.len = item->value.len;
}

int kv_store_init(struct kv_store *s, size_t buckets, size_t
   max_items, bool lru)
{
    s->bucket_count = buckets;
    s->max_items = max_items;
    s->use_lru = lru;
    atomic_set(&s->curr_items, 0);

    s->buckets = kmalloc_array(buckets, sizeof(*s->buckets),
       GFP_KERNEL);
    if (!s->buckets)
        return -ENOMEM;

    for (size_t i = 0; i < buckets; i++) {
        INIT_HLIST_HEAD(&s->buckets[i].head);
    }

    mutex_init(&s->lock);
```

```
    if (s->use_lru)
        lru_init(&s->lru);

    return 0;
}

void kv_store_destroy(struct kv_store *s)
{
    size_t i;
    struct kv_item *item;
    struct hlist_node *tmp;

    mutex_lock(&s->lock);
    for (i = 0; i < s->bucket_count; i++) {
        hlist_for_each_entry_safe(item, tmp,
           &s->buckets[i].head, hnode) {
            hlist_del(&item->hnode);
            kfree(item);
        }
    }

    kfree(s->buckets);
    mutex_unlock(&s->lock);
}

static struct kv_item *kv_bucket_find_item(struct kv_bucket *b,
   struct kv_key *key)
{
    struct kv_item *item = NULL;
    hlist_for_each_entry(item, &b->head, hnode) {
        if (kv_key_equal(&item->key, key)) {
            break;
        }
    }

    return item;
}

static struct kv_item *kv_check_and_evict_item(struct kv_store
   *s)
{
```

```
    struct kv_item *victim;

    if (s->use_lru && atomic_read(&s->curr_items) >=
       s->max_items) {
        victim = lru_evict(&s->lru);
        if (victim) {
            hlist_del_init(&victim->hnode);
        }
    }

    return victim;
}

int kv_put(struct kv_store *s, struct kv_pair *p)
{
    if (s == NULL || p == NULL || p->key.len > KV_MAX_KEY
        || p->value.len > KV_MAX_VAL)
        return -EINVAL;

    u32 hash = kv_key_hash(&p->key) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];
    struct kv_item *item;

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, &p->key);
    if (item) {
        kv_item_set_value(item, p);
        if (s->use_lru)
            lru_touch(&s->lru, item);
        mutex_unlock(&s->lock);
        return 0;
    }

    item = kv_check_and_evict_item(s);
    if (!item) {
        item = kmalloc(sizeof(*item), GFP_KERNEL);
        if (!item) {
            mutex_unlock(&s->lock);
            return -ENOMEM;
        }
        atomic_inc(&s->curr_items);
```

```
    }

    kv_item_init(item, p);
    hlist_add_head(&item->hnode, &b->head);
    if (s->use_lru) {
        INIT_LIST_HEAD(&item->lru_node);
        lru_touch(&s->lru, item);
    }
    mutex_unlock(&s->lock);

    return 0;
}

int kv_get(struct kv_store *s, struct kv_pair *p)
{
    if (s == NULL || p == NULL || p->key.len > KV_MAX_KEY)
        return -EINVAL;

    int err = 0;
    struct kv_item *item;
    u32 hash = kv_key_hash(&p->key) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, &p->key);
    if (item) {
        kv_pair_set_value(p, item);
        if (s->use_lru)
            lru_touch(&s->lru, item);
    } else {
       err = -ENOENT;
    }
    mutex_unlock(&s->lock);

    return err;
}

int kv_del(struct kv_store *s, struct kv_key *k)
{
    if (s == NULL || k == NULL || k->len > KV_MAX_KEY)
        return -EINVAL;
```

```
    int err = 0;
    struct kv_item *item;
    u32 hash = kv_key_hash(k) % s->bucket_count;
    struct kv_bucket *b = &s->buckets[hash];

    mutex_lock(&s->lock);
    item = kv_bucket_find_item(b, k);
    if (item) {
        hlist_del(&item->hnode);
        if (s->use_lru)
            lru_remove(&s->lru, item);
        kfree(item);
        atomic_dec(&s->curr_items);
    } else {
        err = -ENOENT;
    }
    mutex_unlock(&s->lock);

    return err;
}

int kv_stat(struct kv_store *s, struct kv_usage_stat *stat)
{
    if (s == NULL || stat == NULL)
        return -EINVAL;

    stat->bucket_count = s->bucket_count;
    stat->max_items = s->max_items;
    stat->use_lru = s->use_lru;
    stat->curr_items = atomic_read(&s->curr_items);

    return 0;
}
```

Листинг А.5 – Исходный код файла kv_lru.h

```
#ifndef KV_LRU_H
#define KV_LRU_H

#include "kv_types.h"

void lru_init(struct kv_lru *lru);
```

```
void lru_touch(struct kv_lru *lru, struct kv_item *item);
void lru_remove(struct kv_lru *lru, struct kv_item *item);
struct kv_item *lru_evict(struct kv_lru *lru);

#endif // KV_LRU_H
```

Листинг А.6 – Исходный код файла kv_lru.c

```
#include "kv_lru.h"

void lru_init(struct kv_lru *lru)
{
    INIT_LIST_HEAD(&lru->head);
}

void lru_touch(struct kv_lru *lru, struct kv_item *item)
{
    if (list_empty(&item->lru_node))
        list_add(&item->lru_node, &lru->head);
    else
        list_move(&item->lru_node, &lru->head);
}

void lru_remove(struct kv_lru *lru, struct kv_item *item)
{
    if (!list_empty(&item->lru_node))
        list_del_init(&item->lru_node);
}

struct kv_item *lru_evict(struct kv_lru *lru)
{
    struct kv_item *victim = NULL;

    if (!list_empty(&lru->head)) {
        victim = list_last_entry(&lru->head, struct kv_item,
           lru_node);
        list_del_init(&victim->lru_node);
    }

    return victim;
}
```

Листинг А.7 – Исходный код файла kv_mod.c

```
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "kv_store.h"
#include "kv_ioctl.h"

static int buckets = BUCKETS_DEFAULT;
static int max_items = MAX_ITEMS_DEFAULT;
static bool use_lru = USE_LRU_DEFAULT;

module_param(buckets, int, 0444);
module_param(max_items, int, 0444);
module_param(use_lru, bool, 0444);

struct kv_store store;

static long kv_ioctl(struct file *f, unsigned int cmd, unsigned
   long arg)
{
    int err;
    struct kv_pair p;
    struct kv_key k;
    struct kv_usage_stat stat;

    switch (cmd) {
    case KV_PUT:
        if (copy_from_user(&p, (void __user *)arg, sizeof(p)))
            return -EFAULT;
        pr_info("kv_ioctl: KV_PUT key=%.*s value=%.*s\n",
            (int) p.key.len, p.key.data, (int) p.value.len,
               p.value.data);

        return kv_put(&store, &p);

    case KV_GET:
        if (copy_from_user(&p, (void __user *)arg, sizeof(p)))
            return -EFAULT;

        pr_info("kv_ioctl: KV_GET key=%.*s\n", (int) p.key.len,
           p.key.data);
```

```
        err = kv_get(&store, &p);
        if (err != 0)
            return err;

        return copy_to_user((void __user *)arg, &p, sizeof(p));

    case KV_DEL:
        if (copy_from_user(&k, (void __user *)arg, sizeof(k)))
            return -EFAULT;
        pr_info("kv_ioctl: KV_DEL key=%.*s\n", (int) k.len,
           k.data);

        return kv_del(&store, &k);

    case KV_STAT:
        pr_info("kv_ioctl: KV_STAT");
        err = kv_stat(&store, &stat);
        if (err != 0)
            return err;

        return copy_to_user((void __user *)arg, &stat,
           sizeof(stat));

    default:
        pr_info("kv_ioctl: unknown command %u\n", cmd);
        return -EINVAL;
    }
}

const struct file_operations kv_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = kv_ioctl,
};

static struct miscdevice kv_dev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "kvstore",
    .fops = &kv_fops,
};

static int __init kv_init(void)
```

```
{
    pr_info("kv_kernel: init storage");
    int err = kv_store_init(&store, buckets, max_items, use_lru);
    if (err) {
        pr_err("kv_kernel: init failed");
        return err;
    }
    err = misc_register(&kv_dev);
    if (err) {
        pr_err("kv_kernel: misc_register failed");
        return err;
    }
    pr_info("kv_kernel: init storage OK");

    return 0;
}

static void __exit kv_exit(void)
{
    pr_info("kv_storage: destroy storage");
    misc_deregister(&kv_dev);
    kv_store_destroy(&store);
    pr_info("kv_kernel: destroy storage OK");
}

module_init(kv_init);
module_exit(kv_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Gavrilyuk Vladislav");
```

# ПРИЛОЖЕНИЕ Б
## Исходный код библиотеки

Листинг Б.1 – Исходный код файла kv_lib.h

```
#ifndef KV_LIB_H
#define KV_LIB_H
#include <stdint.h>

#define KV_IOC_MAGIC 'k'
#define KV_MAX_KEY   64
#define KV_MAX_VAL   256

typedef uint8_t kv_u8_t;
typedef uint32_t kv_u32_t;
typedef uint64_t kv_u64_t;

struct kv_key {
    char data[KV_MAX_KEY];
    kv_u32_t len;
};

struct kv_value {
    char data[KV_MAX_VAL];
    kv_u32_t len;
};

struct kv_pair {
    struct kv_key key;
    struct kv_value value;
};

struct kv_usage_stat {
    kv_u64_t bucket_count;
    kv_u64_t max_items;
    kv_u64_t cur_items;
    kv_u8_t  use_lru;
};

int kv_open(void);
int kv_close(int fd);
```

```
int kv_put(int fd, const struct kv_pair *pair);
int kv_get(int fd, struct kv_pair *pair);
int kv_del(int fd, const struct kv_key *key);
int kv_stat(int fd, struct kv_usage_stat *stat_out);
const char *kv_err_msg(int err);

#define KV_PUT _IOW(KV_IOC_MAGIC, 1, struct kv_pair)
#define KV_GET _IOWR(KV_IOC_MAGIC, 2, struct kv_pair)
#define KV_DEL _IOW(KV_IOC_MAGIC, 3, struct kv_key)
#define KV_STAT _IOR(KV_IOC_MAGIC, 4, struct kv_usage_stat)

#endif // KV_IOCTL_H
```

Листинг Б.2 – Исходный код файла kv_lib.c

```
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <errno.h>
#include "kv_lib.h"

int kv_open(void)
{
    return open("/dev/kvstore", O_RDWR);
}

int kv_close(int fd)
{
    return close(fd);
}

int kv_put(int fd, const struct kv_pair *pair)
{
    return ioctl(fd, KV_PUT, pair);
}

int kv_get(int fd, struct kv_pair *pair)
{
    return ioctl(fd, KV_GET, pair);
}

int kv_del(int fd, const struct kv_key *key)
{
```

```
    return ioctl(fd, KV_DEL, key);
}

int kv_stat(int fd, struct kv_usage_stat *stat)
{
    return ioctl(fd, KV_STAT, stat);
}

const char *kv_err_msg(int err)
{
    switch (err) {
        case 0:
            return "Success";
        case -ENOENT:
            return "Key not found";
        case -EINVAL:
            return "Invalid argument";
        case -ENOSPC:
            return "No space left in store";
        case -ENOMEM:
            return "Out of memory";
        case -EFAULT:
            return "Bad user pointer";
        default:
            return "Unknown error";
    }
}
```

# ПРИЛОЖЕНИЕ В
## Исходный код тестовых приложений

Листинг В.1 – Исходный код тестового приложения №1

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include "kv_lib.h"
#include <inttypes.h>

void print_stat(int fd)
{
    struct kv_usage_stat stat = {0};
    if (kv_stat(fd, &stat) < 0) {
        perror("kv_stat");
    }

    printf("STAT:\n\tbucket count: %" PRIu64 "\n\tmax_items: %"
       PRIu64
        "\n\tcurr_items: %" PRIu64 "\n\tlru: %s\n",
        stat.bucket_count, stat.max_items, stat.cur_items,
        stat.use_lru ? "on" : "off");
}

int main(void)
{
    int err;
    int fd = kv_open();
    if (fd < 0) {
        perror("kv_open");
        return 1;
    }

    // Fill storage
    print_stat(fd);
    for (int i = 1; i <= 11; i++) {
        struct kv_pair p;

        snprintf(p.key.data, sizeof(p.key), "key%d", i);
        p.key.len = strlen(p.key.data)+1;
        snprintf(p.value.data, sizeof(p.value), "value%d", i);
```

```
        p.value.len = strlen(p.value.data)+1;

        err = kv_put(fd, &p);
        if (err < 0) {
            perror("kv_put");
            printf("%s\n", kv_err_msg(-errno));
        } else {
            printf("PUT: %s -> %s\n", p.key.data, p.value.data);
        }
    }

    print_stat(fd);

    // Check key1
    struct kv_pair p;
    strcpy(p.key.data, "key1\0");
    p.key.len = 5;

    err = kv_get(fd, &p);
    if (err == 0) {
        printf("ERROR: key1 still present: %s\n", p.value.data);
    } else {
        printf("err: %s\n", kv_err_msg(-errno));
        printf("OK: key1 was evicted\n");
    }

    // 3. Get key2
    strcpy(p.key.data, "key2\0");
    err = kv_get(fd, &p);
    if (err == 0) {
        printf("GET: key2 -> %s\n", p.value.data);
    } else {
        printf("ERROR: key2 not found: %s\n",
           kv_err_msg(-errno));
    }

    // 4. Delete key2
    err = kv_del(fd, &p.key);
    if (err == 0) {
        printf("DEL: key2\n");
    } else {
```

```
        printf("err: %s\n", kv_err_msg(-errno));
        perror("kv_del key2");
    }

    // 5. Check key2
    err = kv_get(fd, &p);
    if (err) {
        err = -errno;
        if (err == -ENOENT)
            printf("OK: key2 correctly deleted\n");
        else
            printf("err: %s\n", kv_err_msg(err));
    } else {
        printf("ERROR: key2 still exists\n");
    }

    print_stat(fd);

    kv_close(fd);
    return 0;
}
```

Листинг В.2 – Исходный код тестового приложения №2

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <inttypes.h>
#include <stdlib.h>
#include "kv_lib.h"

#define NUM_KEYS 3
#define NUM_READERS 4
#define NUM_ITER 4

void print_stat(int fd)
{
    struct kv_usage_stat stat = {0};
    if (kv_stat(fd, &stat) < 0) {
        perror("kv_stat");
```

```
    }

    printf("[tid=%ld] STAT:\n\tbucket count: %" PRIu64
       "\n\tmax_items: %" PRIu64
        "\n\tcurr_items: %" PRIu64 "\n\tlru: %s\n",
        pthread_self(), stat.bucket_count, stat.max_items,
        stat.cur_items, stat.use_lru ? "on" : "off");
}

void *writer_thread(void *arg) {
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = 100000000;

    int fd = kv_open();
    if (fd == -1) {
        perror("kv_open");
        return NULL;
    }

    struct kv_pair p;
    for (int i = 0; i < NUM_ITER; i++) {
        for (int k = 0; k < NUM_KEYS; k++) {
            snprintf(p.key.data, KV_MAX_KEY, "key%d", k);
            p.key.len = strlen(p.key.data)+1;
            snprintf(p.value.data, KV_MAX_VAL, "val%d_%d", k, i);
            p.value.len = strlen(p.value.data)+1;
            int err = kv_put(fd, &p);
            if (err != 0) {
                err = -errno;
                printf("WRITER [tid=%ld] PUT ERR: %s -> %s:
                   %s\n", pthread_self(),
                    p.key.data, p.value.data, kv_err_msg(err));
            } else {
                printf("WRITER [tid=%ld] PUT OK: %s -> %s\n",
                   pthread_self(),
                    p.key.data, p.value.data);
            }
            nanosleep(&ts, NULL);
        }
    }
```

```
    print_stat(fd);

    kv_close(fd);
    return NULL;
}

void *reader_thread(void *arg) {
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = 300000000;

    int fd = kv_open();
    if (fd == -1) {
        perror("kv_open");
        return NULL;
    }

    struct kv_pair p;
    for (int k = 0; k < NUM_KEYS; k++) {
        snprintf(p.key.data, KV_MAX_KEY, "key%d", k);
        p.key.len = strlen(p.key.data)+1;
        int err = kv_get(fd, &p);
        if (err != 0) {
            err = -errno;
            printf("READER [tid=%ld] GET ERR: %s: %s\n",
               pthread_self(),
                p.key.data, kv_err_msg(err));
        } else {
            printf("READER [tid=%ld] GET OK: %s -> %s\n",
               pthread_self(),
                p.key.data, p.value.data);
        }
        ts.tv_nsec += rand() % 100000;
        nanosleep(&ts, NULL);
    }
    print_stat(fd);

    kv_close(fd);
    return NULL;
}
```

```
int main() {
    pthread_t writer;
    pthread_t readers[NUM_READERS];

    pthread_create(&writer, NULL, writer_thread, NULL);
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_create(&readers[i], NULL, reader_thread, NULL);
    }

    pthread_join(writer, NULL);
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    printf("Test finished\n");
    return 0;
}
```

# ПРИЛОЖЕНИЕ Г
## Makefile

Листинг Г.1 – Makefile для сборки модуля ядра

```
obj-m := kv.o

kv-objs := \
    kv_store.o \
    kv_lru.o \
    kv_mod.o \

KDIR := /lib/modules/$(shell uname -r)/build
PWD  := $(shell pwd)

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

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean

load:
    sudo insmod kv.ko buckets=128 max_items=100 use_lru=1

unload:
    sudo rmmod kv

reload: unload load
```