



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:

*Мониторинг работы slab кэша*

Студент ИУ7-72Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

И.С.Климов  
(И.О.Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

Н.Ю.Рязанова  
(И.О.Фамилия)

2022 г.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ.....</b>	<b>4</b>
1 Аналитическая часть .....	5
1.1 Постановка задачи .....	5
1.2 Описание принципов работы кэша slab .....	5
1.3 Описание и анализ API для работы со slab .....	6
1.3.1 Функция kmalloc .....	8
1.3.2 Функция kfree .....	11
Выводы.....	12
1.4 Анализ способов перехвата функций в ядре .....	12
1.4.1 Модификация таблицы системных вызовов .....	12
1.4.2 Использование сплайсинга .....	12
1.4.3 Использование ftrace .....	13
1.5 Сравнительный анализ способов перехвата функций ядра.....	15
Выводы.....	15
2 Конструкторская часть .....	16
2.1 Последовательность действий .....	16
2.2 Алгоритм перехвата функции.....	17
2.3 Алгоритм защиты от повторного вызова функции .....	18
2.4 Алгоритмы обработчиков функций .....	19
2.5 Структура программного обеспечения .....	22
3 Технологическая часть .....	23
3.1 Выбор языка и среды программирования .....	23
3.2 Перехват системных вызовов .....	23
3.3 Инициализация структуры ftrace_hook.....	24
3.4 Выполнение функции-перехватчика.....	25
3.5 Выполнение функций-обработчиков .....	25
4 Исследовательская часть .....	27
4.1 Примеры работы разработанного модуля .....	27
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>29</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....</b>	<b>30</b>
<b>ПРИЛОЖЕНИЕ А.....</b>	<b>31</b>

## ВВЕДЕНИЕ

Распределитель памяти slab, используемый в Linux, базируется на алгоритме, впервые введенном Джефом Бонвиком для операционной системы SunOS. Распределитель Джефа строится вокруг объекта кэширования. Внутри ядра значительное количество памяти выделяется на ограниченный набор объектов, например, дескрипторы файлов и другие общие структурные элементы. Джеф основывался на том, что количество времени, необходимое для инициализации регулярного объекта в ядре, превышает количество времени, необходимое для его выделения и освобождения. Его идея состояла в том, что вместо того, чтобы возвращать освободившуюся память в общий фонд, оставлять эту память в проинициализированном состоянии для использования в тех же целях. Например, если выделена для mutex, функцию инициализации необходимо выполнить только один раз, когда память впервые выделяется для mutex. Последующие распределения памяти не требуют выполнения инициализации, поскольку она уже имеет нужный статус от предыдущего освобождения и обращения к деструктору.

В Linux распределитель slab использует эти и другие идеи для создания распределителя памяти, который будет эффективно использовать и пространство, и время [1].

Может возникнуть необходимость исследовать потребление памяти, выделяемой slab, процессом для контроля ее использования. Существующий интерфейс, предоставляемый /proc/slabinfo, а также приложением slabtop, позволяет оценить общий размер кэшей slab, но не позволяет отследить использование памяти конкретным процессом.

**Целью работы** является разработка загружаемого модуля ядра, собирающего статистику выделения памяти slab для конкретного процесса.

# 1 Аналитическая часть

## 1.1 Постановка задачи

В соответствии с заданием на курсовую работу, необходимо разработать программное обеспечение, осуществляющее мониторинг работы slab кэша для конкретного процесса и записывающее информацию в системный журнал. Для решения поставленной задачи необходимо:

- 1) проанализировать функции выделения и освобождения памяти;
- 2) проанализировать механизмы перехвата функций ядра;
- 3) разработать алгоритмы перехвата функций, обработчиков функций выделения и освобождения памяти.

## 1.2 Описание принципов работы кэша slab

Рисунок 1.1 иллюстрирует верхний уровень организации структурных элементов slab. На самом высоком уровне находится `cache_chain`, который является связанным списком кэшей slab. Это полезно для алгоритмов best-fit, которые ищут кэш, наиболее соответствующий по размеру нужного распределения (осуществляя итерацию по списку). Каждый элемент `cache_chain` – это ссылка на структуру (называемая кэшем). Это определяет совокупность объектов заданного размера, которые могут использовать

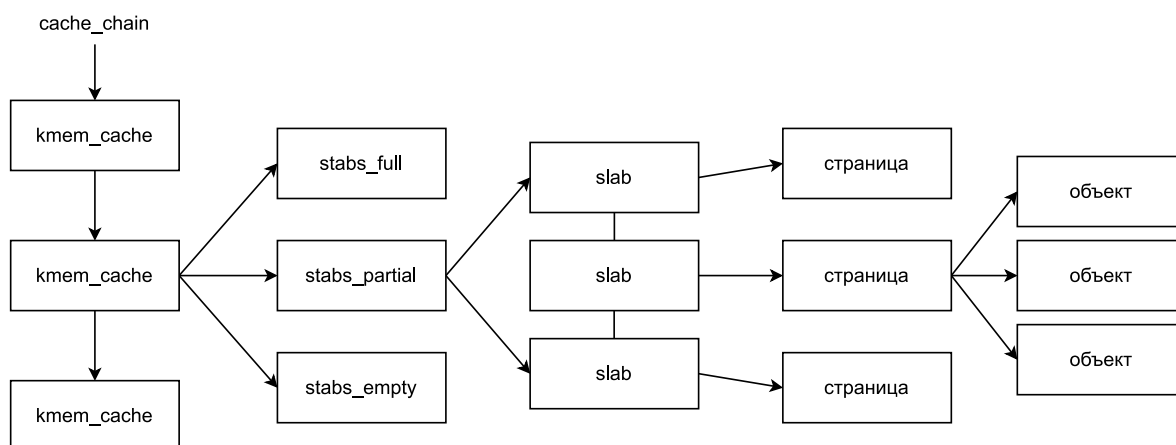


Рисунок 1.1 – Главные структуры распределителя slab

Каждый кэш содержит список slab'ов, которые являются смежными блоками памяти (обычно страницы). Существует три вида slab:

- 1) slabs\_full – slab'ы, которые распределены полностью;
- 2) slabs\_partial – slab'ы, которые распределены частично;
- 3) slabs\_empty – slab'ы, которые являются пустыми, или не выделены под объекты.

В списке slab'ов все slab'ы – смежные блоки памяти (одна или более смежных страниц), которые разделяются между объектами. Эти объекты – основные элементы, которые выделяются из специального кэша и возвращаются в него. Slab – минимальное распределение распределителя slab, поэтому если необходимо увеличить его, это минимум, на который он может увеличиться. Обычно через slab происходит распределение множества объектов.

Поскольку объекты распределяются и освобождаются из slab, отдельные slab могут помещаться между списками slab'ов. Например, когда все объекты в slab израсходованы, они перемещаются из списка slabs\_partial в список slabs\_full. Когда slab полон, и объект освобождается, он перемещается из списка slabs\_full в список slabs\_partial. Когда освобождаются все объекты, они перемещаются из списка slabs\_partial в список slabs\_empty [1].

### **1.3 Описание и анализ API для работы со slab**

Основной структурой кэша slab является struct kmem\_cache. Данная структура содержит описание конкретного кэша. Указатель на эту структуру используется другими функциями кэша slab для создания данного кэша, выделения и освобождения в нем памяти и т.д. Структура kmem\_cache содержит данные, относящиеся к конкретным CPU-модулям, набор настроек (доступных через файловую систему proc), статистических данных и элементов, необходимых для управления кэшем slab [1].

Рассмотрим конкретные функции для работы с кэшем slab. Функция `kmem_cache_create` применяется для создания нового кэша slab и возвращает указатель на этот кэш.

```
struct kmem_cache* kmem_cache_create(  
    const char *name,           // название кэша  
    size_t size,                // выделяемых в кэше объектов  
    size_t align,              // выравнивание объектов  
    unsigned long flags,        // флаги slab  
  
    // callback-функция, вызываемая при выделении объекта  
    void (*ctor)(void*, struct kmem_cache *, unsigned long),  
  
    // callback-функция, вызываемая при освобождении объекта  
    void (*dtor)(void*, struct kmem_cache *, unsigned long)  
);
```

Неполный список флагов slab (полный список представлен в `linux/gfp.h`):

1. `GFP_USER` – выделить память от имени пользователя, может уснуть.
2. `GFP_KERNEL` – выделить оперативную память ядра, может уснуть.
3. `GFP_ATOMIC` – не может уснуть, может использовать запасные пулы.
4. `GFP_NOIO` – запрет на операции ввода / вывода во время выделения памяти.
5. `GFP_NOWAIT` – не может уснуть.

Функция `kmem_cache_alloc` применяется для выделения памяти из конкретного кэша slab.

```
void* kmem_cache_alloc(  
    struct kmem_cache* cachep, // указатель на структуру кэша  
    gfp_t flags                 // флаги slab  
);
```

Функция `kmem_cache_free` применяется для освобождения ранее выделенных объектов.

```
void kmem_cache_free(  
    struct kmem_cache* cachep,    // указатель на кэш  
    void* objp                    // указатель на объект  
);
```

Функция `kmem_cache_destroy` используется для уничтожения кэша. Обычно это происходит при выгрузке модуля.

```
void kmem_cache_destroy(  
    struct kmem_cache* cachep    // указатель на кэш  
);
```

Как можно заметить, для выделения памяти с помощью этих функций необходимо указывать явно, какой кэш требуется использовать, а, следовательно, контролировать его можно из пространства пользователя с помощью `/proc/slabinfo`, читая информацию о требуемом кэше.

Однако основным способом выделения и освобождения памяти являются `kmalloc` и `kfree`.

```
void* kmalloc(size_t size, gfp_t flags);  
void kfree(const void *objp)
```

В `kmalloc` необходимые для распределения аргументы – только размер объектов и набор флагов. Чтобы убедиться в том, что данные функции работают со `slab` кэшем, рассмотрим реализации каждой из них.

### 1.3.1 Функция `kmalloc`

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)  
{  
    if (__builtin_constant_p(size)) {  
        ...  
    }  
    return __kmalloc(size, flags);  
}
```

Функция начинается с проверки на то, является ли переданный `size` константой. Если да, то память будет выделена из кэша для объектов соответствующего размера, кэш для объектов не будет искаться в процессе исполнения кода, то есть он будет найден в процессе исполнения кода. В противном же случае вызывается функция `__kmalloc`.

```
void *__kmalloc(size_t size, gfp_t flags)
{
    return __do_kmalloc(size, flags, _RET_IP_);
}

static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,
                                           unsigned long caller)
{
    struct kmem_cache *cachep;
    void *ret;

    if (unlikely(size > KMALLOC_MAX_CACHE_SIZE))
        return NULL;
    cachep = kmalloc_slab(size, flags);
    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
        return cachep;
    ret = slab_alloc(cachep, flags, caller);

    ret = kasan_kmalloc(cachep, ret, size, flags);
    trace_kmalloc(caller, ret,
                  size, cachep->size, flags);

    return ret;
}
```

Все, что делает `__kmalloc` – вызывает `__do_malloc`, которая и выполняет всю логику. Особый интерес в ней представляют два вызова – `kmalloc_slab` и `slab_alloc`.

1) Первая функция – находит структуру `kmem_cache` в `slab`'е, которая соответствует необходимому размеру. При этом, в зависимости от величины, расчет индекса происходит по-разному.



```

struct kmem_cache *kmalloc_slab(size_t size, gfp_t flags)
{
    unsigned int index;

    if (size <= 192) {
        if (!size)
            return ZERO_SIZE_PTR;

        index = size_index[size_index_elem(size)];
    } else {
        if (WARN_ON_ONCE(size > KMALLOC_MAX_CACHE_SIZE))
            return NULL;
        index = fls(size - 1);
    }

    return kmalloc_caches[kmalloc_type(flags)][index];
}

```

2) Вторая функция – выделяет память из slab'а и подготавливает ее.

```

static __always_inline void *
slab_alloc(struct kmem_cache *cachep, gfp_t flags, unsigned long caller)
{
    unsigned long save_flags;
    void *objp;

    flags &= gfp_allowed_mask;
    cachep = slab_pre_alloc_hook(cachep, flags);
    if (unlikely(!cachep))
        return NULL;

    cache_alloc_debugcheck_before(cachep, flags);
    local_irq_save(save_flags);
    objp = __do_cache_alloc(cachep, flags);
    local_irq_restore(save_flags);
    objp = cache_alloc_debugcheck_after(cachep, flags, objp, caller);
    prefetchw(objp);

    if (unlikely(slab_want_init_on_alloc(flags, cachep)) && objp)
        memset(objp, 0, cachep->object_size);

    slab_post_alloc_hook(cachep, flags, 1, &objp);
    return objp;
}

```

Поиск памяти для кэша происходит в функции `memcg_kmem_get_cache`, которая вызывается из `slab_pre_alloc_hook`. В случае отсутствия такого – вызывается функция создания кэша.

```
static inline struct kmem_cache *slab_pre_alloc_hook(
    struct kmem_cache *s,
    gfp_t flags)
{
    ...

    if (memcg_kmem_enabled() &&
        ((flags & __GFP_ACCOUNT) || (s->flags & SLAB_ACCOUNT)))
        return memcg_kmem_get_cache(s);

    return s;
}
```

```
struct kmem_cache *memcg_kmem_get_cache(struct kmem_cache *cachep)
{
    ...

    if (unlikely(!memcg_cachep))
        memcg_schedule_kmem_cache_create(memcg, cachep);
    else if (percpu_ref_tryget(&memcg_cachep->memcg_params.refcnt))
        cachep = memcg_cachep;

    ...
}
```

Таким образом, можно видеть, что функция `kmalloc` обращается за памятью к `slab` кэшу. Если память для подходящего кэша не выделена, то происходит ее выделение, и данный кэш помечается как используемый. Стоит отметить, что для мониторинга можно отслеживать функцию `__kmalloc`. По окончании использования памяти необходимо ее освободить (сообщить `slab`'у о том, что память больше не используется). За это отвечает функция `kfree`.

### 1.3.2 Функция `kfree`

```
void kfree(const void *objp)
{
    ...

    c = virt_to_cache(objp);

    ...

    __cache_free(c, (void *)objp, _RET_IP_);

    ...
}
```

Для освобождения памяти, выделенной функцией `kmalloc`, кэш, из которого был выделен объект, определяется вызовом `virt_to_cache`. Эта функция возвращает ссылку на кэш, которая затем используется в запросе к `__cache_free` для освобождения объекта.

## **Выводы**

Таким образом, `kmalloc` и `kfree` используют кэш slab точно так же, как и определенные ранее функции. Задача мониторинга выделения памяти с помощью `kmalloc` не может быть решена с помощью стандартного системного интерфейса, следовательно, разрабатываемый модуль должен решить эту задачу. Необходимо отслеживать системные вызовы данных функций и оценивать количество выделенной памяти для исследуемого процесса.

### **1.4 Анализ способов перехвата функций в ядре**

#### **1.4.1 Модификация таблицы системных вызовов**

Для перехвата функций, присутствующих в таблице системных вызовов `sys_call_table`, существует возможность заменить строку в данной таблице на соответствующую функцию с совпадающей сигнатурой, которая выполнит сбор статистики, а затем произведет вызов оригинальной функции и вернет ее результат. К сожалению, функции, используемые для работы со slab, не присутствуют в данной таблице, так что этот способ не подходит для решения поставленной задачи.

#### **1.4.2 Использование сплайсинга**

Классический способ перехвата функций. Инструкции в начале функции заменяются на безусловный переход в новый обработчик. Оригинальные

функции переносятся в другое место и исполняются перед переходом обратно в функцию [2].

Данный способ работает для любой функции, при условии, что ее адрес известен. Однако, он сопряжен с рядом сложностей.

1. Необходимость синхронизации установки и снятия перехвата (для случаев, когда функция будет вызвана в момент установки перехвата).
2. Необходимость обхода защиты на модификацию регионов памяти с кодом.
3. Проверка на отсутствие переходов в заменяемый кусок кода.

Данный подход является эффективным, но существует встроенный в систему фреймворк, решающий данную задачу за программиста, что обеспечивает более высокий уровень надежности решения.

### **1.4.3 Использование ftrace**

Ftrace – фреймворк для трассировки функций, встроенный в ядро Linux с версии 2.6.27 [3]. Реализуется на основе ключей компилятора `-pg` и `-mfentry`, которые вставляют в начало каждой функции вызов специальной трассировочной функции `mcount()` или `__fentry__()`. Большинство популярных дистрибутивов Linux компилируются с этими ключами. Обычно, в пользовательских программах эта возможность компилятора используется профилировщиком, чтобы отследить вызовы всех функций. Ядро же использует эти функции для реализации ftrace. Ftrace работает динамически. Он знает места расположения всех функций `mcount()` и `__fentry__()` и по умолчанию заменяет их машинный код на `nop` – пустую инструкцию. Таким образом, ftrace практически не замедляет работу системы.

Обработчик описывается структурой `ftrace_ops`, из которых нас интересуют два поля:

```

struct ftrace_ops(
    .func,    // callback-функция
    .flags    // флаги
);

```

Регистрация и deregистрация обработчик производится с помощью функций:

- `int register_ftrace_function(struct ftrace_ops *ops);`
- `int unregister_ftrace_function(struct ftrace_ops *ops);`

Данные функции принимают указатель на ранее описанную структуру.

Callback-функция имеет следующий вид:

```

void callback_func(
    unsigned long ip,           // IP трассируемой функции
    unsigned long parent_ip,    // IP функции, вызвавшей
                                // трассируемую функцию

    struct ftrace_op* op,       // указатель на структуру, с помощью
                                // которой была произведена
                                // регистрация обработчика

    struct pt_regs             // структура, позволяющая устанавливать
                                // значения в регистрах процессора после
                                // выхода из callback-а, если был установлен
                                // соответствующий флаг
)

```

Для получения IP трассируемой функции можно использовать функцию `unsigned long kallsyms_lookup_name(const char* name)`, которая возвращает адрес функции по ее названию.

Для перехвата функции, необходимо в callback-функции изменить поле `ip` структуры `regs` на адрес функции. Нужно предотвратить рекурсивный перехват оригинальной функции при ее вызове из обработчика. Для этого используется функция `int within_module(unsigned long addr, const struct module* mod)`, в которую передаются параметр `parent_ip` и макрос `THIS_MODULE`. Если функция возвращает нулевое значение, то вызов произошел из этого модуля, и редактировать IP не нужно.

## 1.5 Сравнительный анализ способов перехвата функций ядра

Сравнение методов перехвата функций в ядре представлено в таблице 1.1.

Таблица 1 – Сравнение способов перехвата функции в ядре

Критерий Способ	Перехват функций для работы со slab	Низкие требования к ядру	Низкая техническая сложность	Перехват функций по имени
<b>Модификация</b>	—	+	+	—
<b>Сплайсинг</b>	+	+	—	—
<b>Ftrace</b>	+	+	+	+

### Выводы

В соответствии с проведенным анализом функций в состав программного обеспечения будет входить модуль ядра ОС Linux, отслеживающий вызов функций `kmalloc` и `kfree` для заданного процесса, считающий статистику выделения и освобождения памяти и записывающий ее в системный журнал. Для выполнения поставленной задачи необходимо уметь осуществлять перехват функций ядра, работающих со slab кэшем. В результате сравнения способов перехвата выявлено, что наиболее эффективным является использование фреймворка `ftrace`, который обладает удобным интерфейсом, имеет низкую техническую сложность и позволяет перехватывать функцию по имени.

## 2 Конструкторская часть

### 2.1 Последовательность действий

На рисунке 2.1 представлена IDEF0-диаграмма нулевого уровня для разрабатываемой программы.

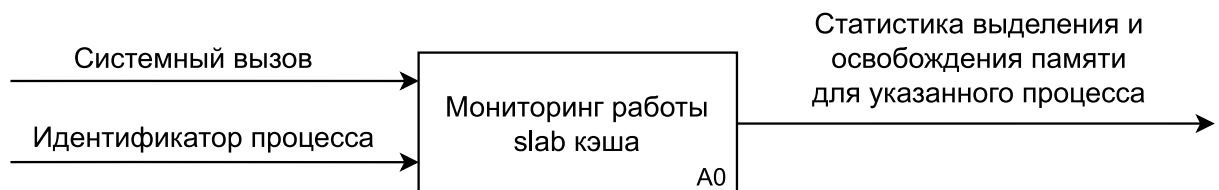


Рисунок 2.1 – IDEF0-диаграмма нулевого уровня

Загружаемый модуль ядра должен обеспечить выполнение последовательности действий, представленных на рисунке 2.2.

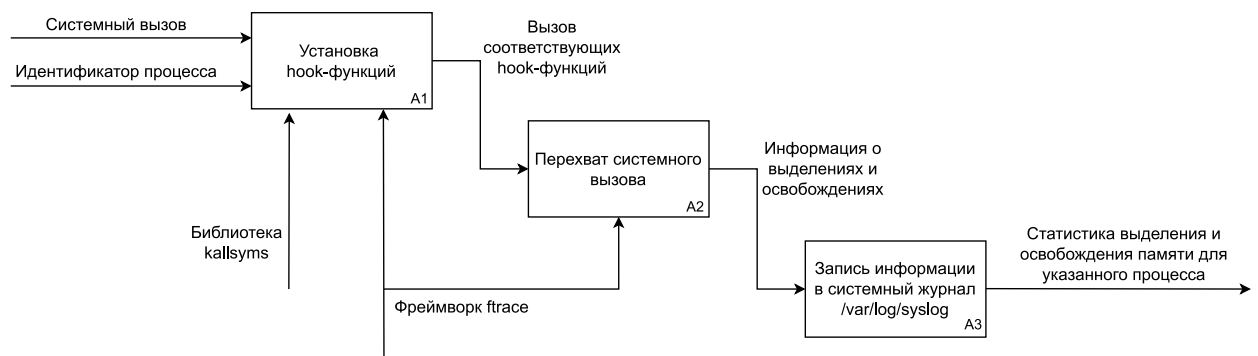


Рисунок 2.2 – IDEF0-диаграмма первого уровня

На первом этапе необходимо установить hook-функции, собирающие статистику о выделенной и освобожденной памяти. Затем происходит перехват системного и получение статистики об использованной памяти. Третий этап – запись полученной информации в системный журнал /var/log/syslog.

## 2.2 Алгоритм перехвата функции

Алгоритм перехвата функции представлен на рисунке 2.3.

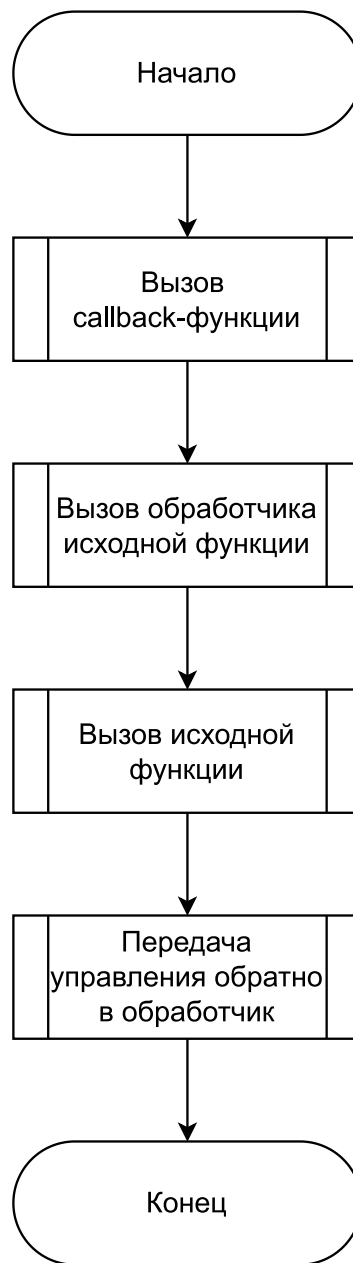


Рисунок 2.3 – Алгоритм перехвата функции

После очередного системного вызова происходит вызов callback-функции, которая передает управление функции-обработчику. Внутри нее вызывается исходная функция, а затем сохраняется необходимая статистика.



## 2.3 Алгоритм защиты от повторного вызова функции

В результате работы обработчика происходит повторный вызов функции ядра, в результате чего появляется рекурсивная обработка. Для предотвращения этого необходимо делать проверку адреса вызываемой стороны, если он произведен вне модуля, то управление передается, иначе – ничего не происходит. Алгоритм защиты представлен на рисунке 2.4.



Рисунок 2.4 – Алгоритм защиты от повторного вызова функции

## **2.4 Алгоритмы обработчиков функций**

В соответствии с требованиями, необходимо составить алгоритмы обработчиков двух функций – `kmalloc` и `kfree`, схемы данных алгоритмов представлены на рисунках 2.5 и 2.6 соответственно. В начале каждого из них необходимо вызвать оригинальную функцию, затем определить идентификатор текущего процесса. Если он равен -1, то выводится информация о процессе в результате выполнения функции. Если же совпадает с отслеживаемым PID, то помимо этого сохраняется количество выделенной памяти с момента загрузки модуля и поддерживается актуальное состояние счетчика.

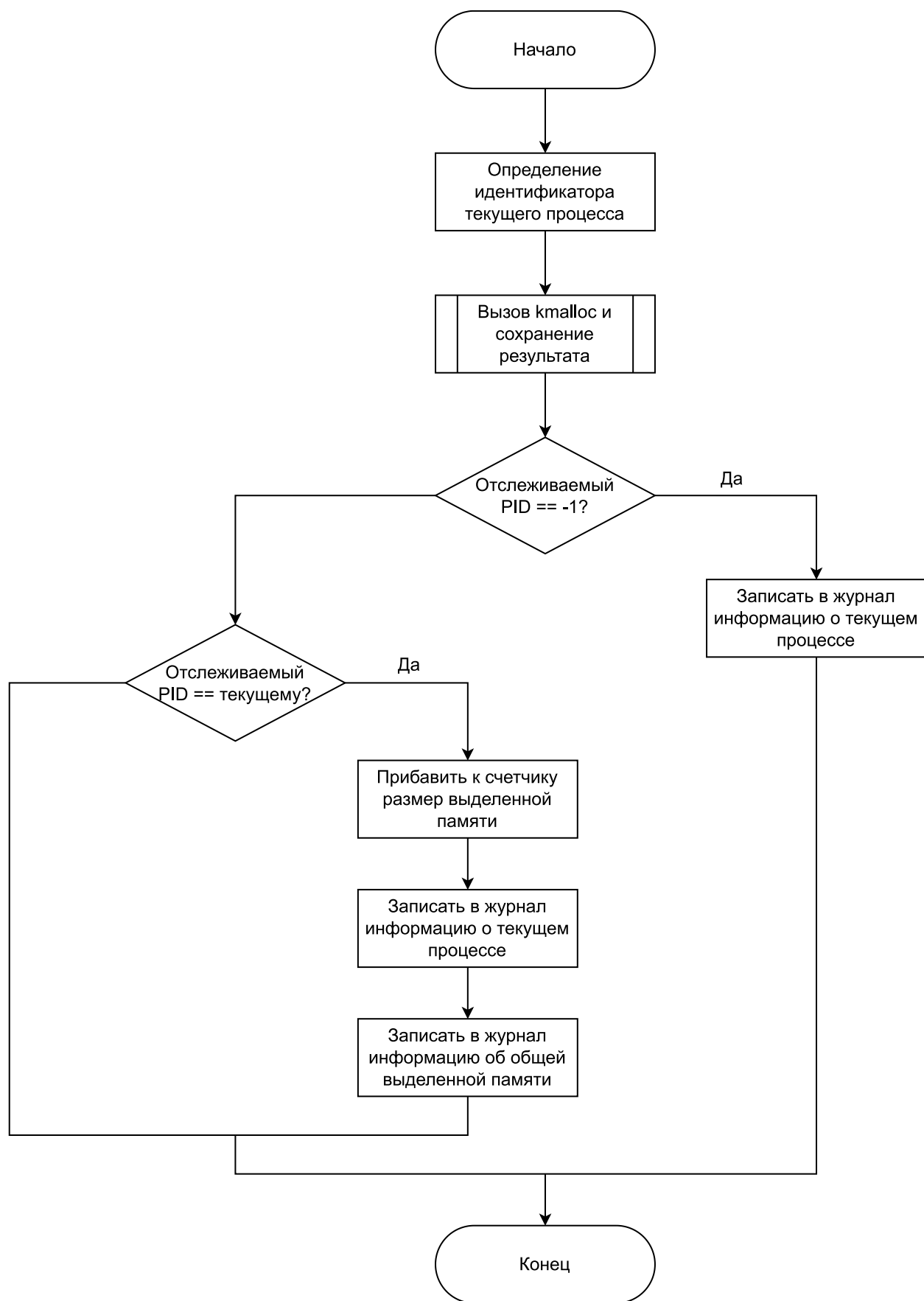


Рисунок 2.5 – Алгоритма обработчика kmalloc

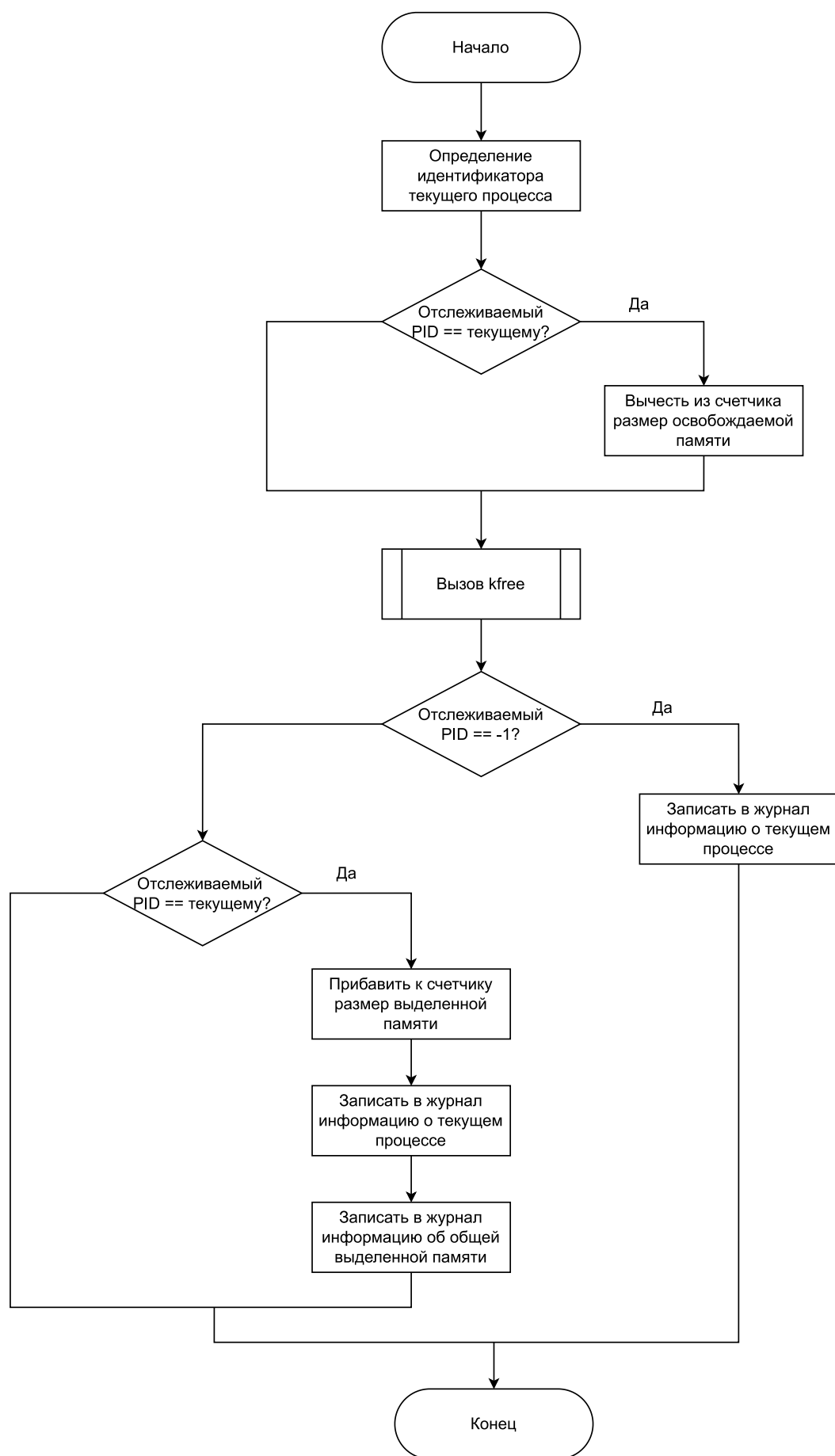


Рисунок 2.6 – Алгоритм обработчика kfree

## 2.5 Структура программного обеспечения

Разрабатываемый модуль должен обеспечивать регистрацию новых обработчиков функций `kmalloc()` и `kfree()` при инициализации, сбор статистики по их использованию для заданного процесса, вызов оригинальных функций и возвращение их результата, а также восстановление стандартного режима функционирования функций `kmalloc()` и `kfree()` при выгрузке модуля. Таким образом обеспечивается необходимая функциональность и продолжение нормальной работы устройства. Структура программного обеспечения представлена на рисунке 2.7.

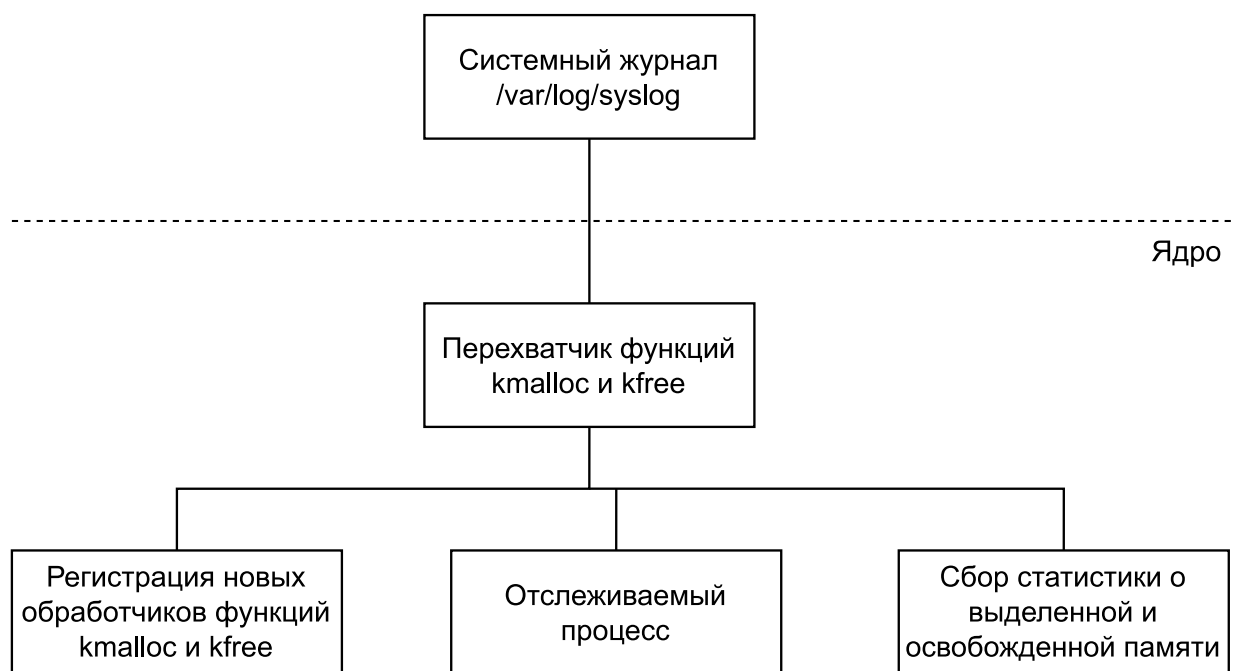


Рисунок 2.7 – Структурная схема ПО

## 3 Технологическая часть

### 3.1 Выбор языка и среды программирования

Наиболее оптимальным выбором языка программирования для написания загружаемого модуля является язык С. Для компиляции модуля используется компилятор gcc, для сборки – утилита make, среда разработки – Notepad3 (быстрый и легковесный текстовый редактор с подсветкой синтаксиса).

### 3.2 Перехват системных вызовов

Информация, которая необходима ftrace для перехвата, представлена в структуре ftrace\_hook. Первые три поля заполняются вручную, остальные – в самой реализации.

```
struct ftrace_hook {  
    const char* name;           // имя перехватываемой функции  
    void* function;             // указатель на новую функцию  
    void* original;             // указатель на оригинальную функцию  
  
    unsigned long address;       // адрес оригинальной функции  
    struct ftrace_ops ops;       // структура для работы с ftrace  
};
```

Для описания функции используется макрос HOOK, который содержит имя перехватываемой функции, указатели на новую и оригинальную функции.

```
#define HOOK(_name, _function, _original) { \  
    .name = (_name), \  
    .function = (_function), \  
    .original = (_original), \  
}
```

Также для удобства все перехватываемые функции помещаются в массив структур ftrace\_hook, который располагается в глобальной области видимости.

```
static struct ftrace_hook slab_hooks[] = {
    HOOK("__kmalloc", fh_kmalloc, &real_kmalloc),
    HOOK("kfree", fh_kfree, &real_kfree),
};
```

### 3.3 Инициализация структуры ftrace\_hook

В первую очередь необходимо найти и сохранить адрес перехватываемой функции. Это можно сделать при помощи функции `kallsyms_lookup_name`.

```
static int fh_resolve_hook_address(struct ftrace_hook* hook) {
    hook->address = kallsyms_lookup_name(hook->name);
    if (!hook->address) {
        pr_debug("Invalid name: %s\n", hook->name);
        return -ENOENT;
    }

    *((unsigned long*) hook->original) = hook->address;
    return 0;
}
```

Далее нужно инициализировать структуру `ftrace_ops` путем заполнения поля `func` callback-функцией, а также выставления флагов (поле `flags`). Затем можно вызвать функции, необходимые для начала работы самого фреймворка.

```
int fh_install_hook(struct ftrace_hook *hook) {
    int err = fh_resolve_hook_address(hook);
    if (err)
        return err;
    hook->ops.func = fh_ftrace_thunk; // установка callback

    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION_SAFE
                    | FTRACE_OPS_FL_IPMODIFY;

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err)
        return err;
    err = register_ftrace_function(&hook->ops);

    if (err) {
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }

    return 0;
}
```

Для выключения перехвата необходимо выполнить действия в обратном порядке.

```
void fh_remove_hook(struct ftrace_hook* hook) {
    int err = unregister_ftrace_function(&hook->ops);
    if (err)
        pr_debug("unregister_ftrace_function() failed: %d\n", err);

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err)
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
}
```

### 3.4 Выполнение функции-перехватчика

Ftrace позволяет изменять состояния регистров, поэтому мы можем поменять адрес следующей выполняемой инструкции и таким образом осуществить переход в написанную функцию.

```
static void notrace fh_ftrace_thunk(unsigned long ip,
                                     unsigned long parent_ip,
                                     struct ftrace_ops* ops,
                                     struct pt_regs* regs) {
    // Получение структуры по значению поля с помощью макроса
    struct ftrace_hook* hook = container_of(ops, struct ftrace_hook, ops);

    // Установка ip на новую функцию, если вызов произведен вне модуля
    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long) hook->function;
}
```

### 3.5 Выполнение функций-обработчиков

Если указана отслеживаемый процесс, то обработчики функций kmalloc и kfree должны поддерживать актуальное состояние переменной, отвечающей за общее число выделенной памяти, и записать эту информацию в системный журнал для указанного процесса. Помимо этого, должна записываться информация об идентификаторе процесса, количество запрашиваемых байт, флаги (в случае kfree – адрес освобождаемой памяти) и код возврата.



```

static void* fh_kmalloc(size_t size, gfp_t flags) {
    const pid_t pid = task_pid_nr(current); // определение текущего pid
    void* ret = real_kmalloc(size, flags);

    if (tracing_pid == -1) {
        pr_info("pid %u called kmalloc(%lu, %X) and got address %8p\n",
                task_pid_nr(current), size, flags, ret);
    } else if (pid == tracing_pid) { // вывод для конкретного pid
        if (ret != NULL)
            total_memory_bytes += ksize(ret);

        pr_info("pid %u called kmalloc(%lu, %X) and got adress %8p\n",
                tracing_pid, size, flags, ret);
        pr_info("Total memory allocated since module init: %ldB\n",
                total_memory_bytes);
    }
    return ret;
}

```

```

static void fh_kfree(const void* objp) {
    if (objp == NULL)
        return;

    const pid_t pid = task_pid_nr(current);
    if (tracing_pid == pid)
        total_memory_bytes -= ksize(objp);

    real_kfree(objp);

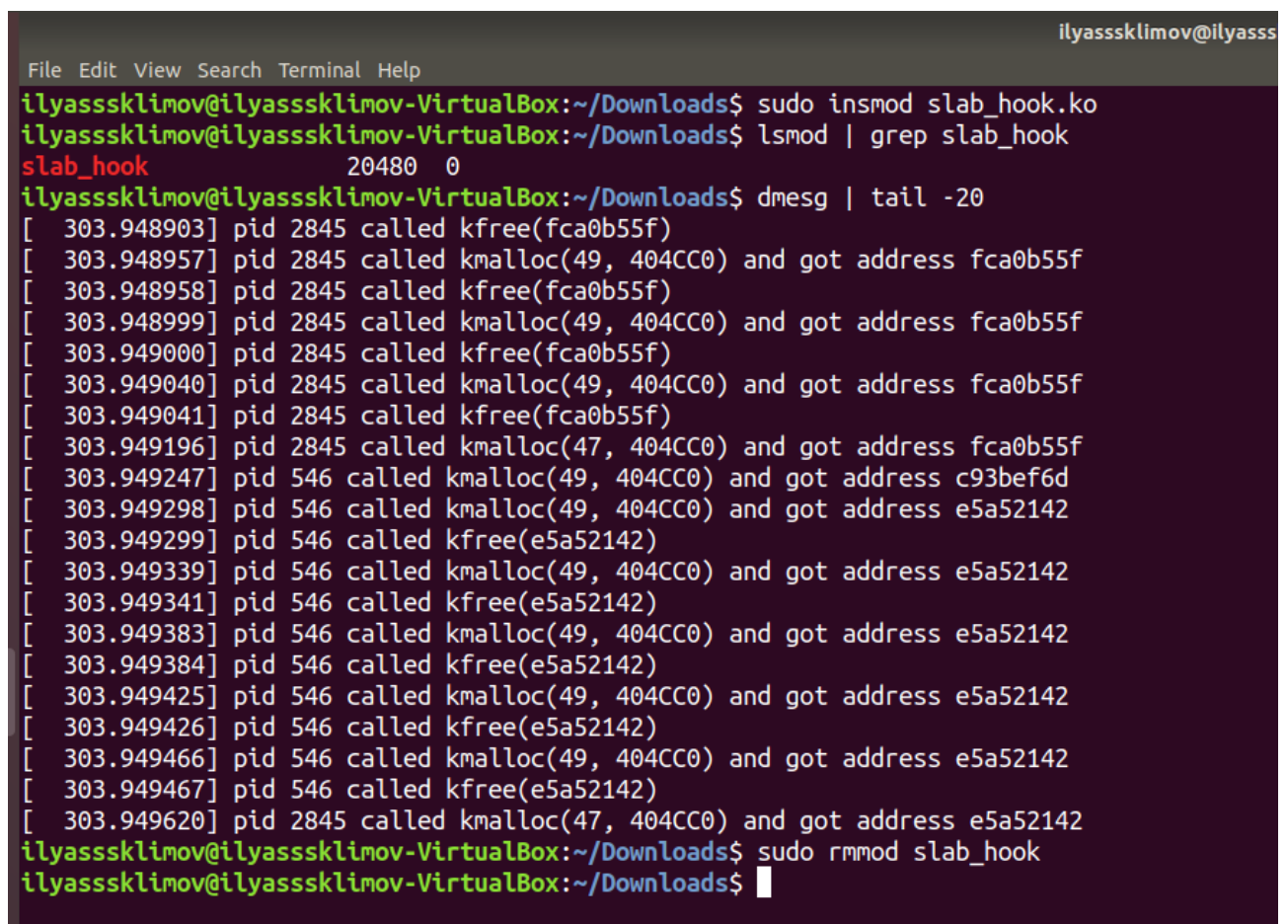
    if (tracing_pid == -1) {
        pr_info("pid %u called kfree(%8p)\n", task_pid_nr(current), objp);
    } else if (tracing_pid == pid) {
        pr_info("pid %u called kfree(%8p)\n", tracing_pid, objp);
        pr_info("Total memory allocated since module init: %ldB\n",
                total_memory_bytes);
    }
}

```

## 4 Исследовательская часть

### 4.1 Примеры работы разработанного модуля

На рисунке 4.1 показан результат работы модуля после загрузки без указания процесса. Как видно, происходит постоянное выделение и освобождение памяти, выводится указанная информация.



```
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ sudo insmod slab_hook.ko
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ lsmod | grep slab_hook
slab_hook                20480  0
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ dmesg | tail -20
[ 303.948903] pid 2845 called kfree(fca0b55f)
[ 303.948957] pid 2845 called kmalloc(49, 404CC0) and got address fca0b55f
[ 303.948958] pid 2845 called kfree(fca0b55f)
[ 303.948999] pid 2845 called kmalloc(49, 404CC0) and got address fca0b55f
[ 303.949000] pid 2845 called kfree(fca0b55f)
[ 303.949040] pid 2845 called kmalloc(49, 404CC0) and got address fca0b55f
[ 303.949041] pid 2845 called kfree(fca0b55f)
[ 303.949196] pid 2845 called kmalloc(47, 404CC0) and got address fca0b55f
[ 303.949247] pid 546 called kmalloc(49, 404CC0) and got address c93bef6d
[ 303.949298] pid 546 called kmalloc(49, 404CC0) and got address e5a52142
[ 303.949299] pid 546 called kfree(e5a52142)
[ 303.949339] pid 546 called kmalloc(49, 404CC0) and got address e5a52142
[ 303.949341] pid 546 called kfree(e5a52142)
[ 303.949383] pid 546 called kmalloc(49, 404CC0) and got address e5a52142
[ 303.949384] pid 546 called kfree(e5a52142)
[ 303.949425] pid 546 called kmalloc(49, 404CC0) and got address e5a52142
[ 303.949426] pid 546 called kfree(e5a52142)
[ 303.949466] pid 546 called kmalloc(49, 404CC0) and got address e5a52142
[ 303.949467] pid 546 called kfree(e5a52142)
[ 303.949620] pid 2845 called kmalloc(47, 404CC0) and got address e5a52142
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ sudo rmmod slab_hook
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$
```

Рисунок 4.1 – Результат работы модуля без указания отслеживаемого процесса

В случае, если указать конкретный процесс, то будет также выводиться информация о выделенной памяти. На рисунке 4.2 показан пример работы модуля для данного случая.

```
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ sudo insmod slab_hook.ko
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ lsmod | grep slab_hook
slab_hook                20480  0
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$ dmesg | tail -20
[ 2256.006589] pid 236 called kmalloc(96, D40) and got adress c8098f0
[ 2256.006592] Total memory allocated since module init: -416B
[ 2256.006597] pid 236 called kfree( c8098f0)
[ 2256.006597] Total memory allocated since module init: -512B
[ 2256.506371] pid 236 called kmalloc(96, D40) and got adress c8098f0
[ 2256.506373] Total memory allocated since module init: -416B
[ 2256.506378] pid 236 called kfree( c8098f0)
[ 2256.506378] Total memory allocated since module init: -512B
[ 2257.008869] pid 236 called kmalloc(96, D40) and got adress c8098f0
[ 2257.008871] Total memory allocated since module init: -416B
[ 2257.008876] pid 236 called kfree( c8098f0)
[ 2257.008876] Total memory allocated since module init: -512B
[ 2257.505642] pid 236 called kmalloc(96, D40) and got adress c8098f0
[ 2257.505644] Total memory allocated since module init: -416B
[ 2257.505649] pid 236 called kfree( c8098f0)
[ 2257.505649] Total memory allocated since module init: -512B
[ 2258.005782] pid 236 called kmalloc(96, D40) and got adress e17cc8ba
[ 2258.005784] Total memory allocated since module init: -416B
[ 2258.005790] pid 236 called kfree(e17cc8ba)
[ 2258.005791] Total memory allocated since module init: -512B
ilyasssklimov@ilyasssklimov-VirtualBox:~/Downloads$
```

Рисунок 4.2 – Результат работы модуля с указанием отслеживаемого процесса

## **ЗАКЛЮЧЕНИЕ**

В результате выполнения работы был реализован загружаемый модуль ядра ОС Linux для отслеживания работы slab кэша для конкретного процесса. Данный модуль путем перехвата системных вызовов `kmalloc` и `kfree` выводит информацию о выделенной памяти и информации о вызове. Для перехвата функций используется фреймворк `ftrace`.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Анатомия распределителя памяти slab в Linux, М. Джонс – <https://www.ibm.com/developerworks/ru/library/l-linux-slab-allocator/index.html> (Дата обращения: 20.12.2022)
2. Перехват функций в ядре Linux с помощью ftrace, А. Лозовский – <https://habr.com/post/413241/> (Дата обращения: 20.12.2022)
3. Архив исходных кодов ядра Linux – <https://elixir.bootlin.com> (Дата обращения: 22.12.2022)

# ПРИЛОЖЕНИЕ А

## Код разрабатываемого модуля

```
// Для недопущения ошибок возврата в вызывающую функцию
#pragma GCC optimize("-fno-optimize-sibling-calls")

#include <linux/ftrace.h>
#include <linux/kallsyms.h>
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/sched.h>
#include <linux/gfp.h>

MODULE_DESCRIPTION("Slab cache monitoring module");
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Klimov Ilya") ;

// Описание входного параметра модуля
static pid_t tracing_pid = -1;
module_param(tracing_pid, uint, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(tracing_pid, "This is pid of process which we are
tracing.");

// Переменная, хранящая суммарное количество выделенных байт с момента
загрузки модуля
static long total_memory_bytes = 0;

// =====
// || kmalloc и kfree ||
// =====

// Указатель на настоящий kmalloc
static void* (*real_kmalloc) (size_t size, gfp_t flags);

// Функция, собирающая статистику (kmalloc)
static void* fh_kmalloc(size_t size, gfp_t flags) {
    const pid_t pid = task_pid_nr(current); // определение текущего pid
    void* ret = real_kmalloc(size, flags);

    // Общий вывод
    if (tracing_pid == -1) {
        pr_info("pid %u called kmalloc(%lu, %X) and got address %8p\n",
            task_pid_nr(current), size, flags, ret);
    } else if (pid == tracing_pid) { // вывод для конкретного pid
        if (ret != NULL)
            total_memory_bytes += ksize(ret);

        pr_info("pid %u called kmalloc(%lu, %X) and got adress %8p\n",
            tracing_pid, size, flags, ret);
        pr_info("Total memory allocated since module init: %ldB\n",
```

```

        total_memory_bytes);
    }
    return ret;
}

// Указатель на настоящий kfree
static void (*real_kfree) (const void* objp);

// Функция, собирающая статистику (kfree)
static void fh_kfree(const void* objp) {
    if (objp == NULL)
        return;

    const pid_t pid = task_pid_nr(current);
    if (tracing_pid == pid)
        total_memory_bytes -= ksize(objp);

    real_kfree(objp);

    if (tracing_pid == -1) {
        pr_info("pid %u called kfree(%8p)\n", task_pid_nr(current),
objp);
    } else if (tracing_pid == pid) {
        pr_info("pid %u called kfree(%8p)\n", tracing_pid, objp);
        pr_info("Total memory allocated since module init: %ldB\n",
            total_memory_bytes);
    }
}

// =====
// || HOOK ||
// =====

// Макрос для инициализации структур, описывающих перехватываемые функции
#define HOOK(_name, _function, _original) { \
    .name = (_name), \
    .function = (_function), \
    .original = (_original), \
}

// Структура, описывающая перехватываемую функцию
struct ftrace_hook {
    const char* name; // имя перехватываемой функции
    void* function; // указатель на новую функцию
    void* original; // указатель на оригинальную функцию

    unsigned long address; // адрес оригинальной функции
    struct ftrace_ops ops; // структура для работы с ftrace
};

// Массив необходимых для перехвата структур
static struct ftrace_hook slab_hooks[] = {
    HOOK("__kmalloc", fh_kmalloc, &real_kmalloc),
    HOOK("kfree", fh_kfree, &real_kfree),
};

```

```

// Функция, заполняющая поля структуры ftrace_hook (address,
original)
static int fh_resolve_hook_address(struct ftrace_hook* hook) {
    // Определение адреса оригинальной функции
    hook->address = kallsyms_lookup_name(hook->name);
    if (!hook->address) {
        pr_debug("Invalid name: %s\n", hook->name);
        return -ENOENT;
    }

    // Получение указателя на функцию
    *((unsigned long*) hook->original) = hook->address;
    return 0;
}

// =====
// || Callback-функция ||
// =====

static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long
parent_ip,
                                   struct ftrace_ops* ops, struct
pt_regs* regs) {
    // Получение структуры по значению поля с помощью макроса
    struct ftrace_hook* hook = container_of(ops, struct ftrace_hook,
ops);

    // Установка ip на новую функцию, если вызов произведен вне модуля
    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long) hook->function;
}

// =====
// || Функция, выключающая перехват ||
// =====

void fh_remove_hook(struct ftrace_hook* hook) {
    int err = unregister_ftrace_function(&hook->ops);
    if (err)
        pr_debug("unregister_ftrace_function() failed: %d\n", err);

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err)
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
}

void fh_remove_hooks(struct ftrace_hook* hooks, size_t count) {
    size_t i = 0;
    for (; i < count; i++)
        fh_remove_hook(&hooks[i]);
}

// =====
// || Функция, устанавливающая перехват ||
// =====

```



```

int fh_install_hook(struct ftrace_hook *hook) {
    int err = fh_resolve_hook_address(hook);
    if (err)
        return err;
    hook->ops.func = fh_ftrace_thunk; // установка callback

    // Установка флагов, позволяющих модифицировать регистры
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION_SAFE
                    | FTRACE_OPS_FL_IPMODIFY;

    // Запуск ftrace для функции по адресу hook->address
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err)
        return err;

    err = register_ftrace_function(&hook->ops);
    if (err) {
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }

    return 0;
}

int fh_install_hooks(struct ftrace_hook *hooks, size_t count) {
    int err = 0, hook_failed = 0;
    size_t i = 0;
    for (; i < count; i++) {
        err = fh_install_hook(&hooks[i]);
        if (err) {
            hook_failed = 1;
            break;
        }
    }

    if (hook_failed) {
        while (i != 0)
            fh_remove_hook(&hooks[--i]);
        return err;
    }

    return 0;
}

// =====
// || Загрузка и выгрузка модуля ||
// =====

static int fh_init(void) {
    int err = fh_install_hooks(slab_hooks, ARRAY_SIZE(slab_hooks));
    if (err)
        return err;

    pr_info("Module is loaded. Tracing pid = %u\n", tracing_pid);
    return 0;
}

```

```
static void fh_exit(void) {  
    fh_remove_hooks(slab_hooks, ARRAY_SIZE(slab_hooks));  
    pr_info("Module is unloaded\n");  
}  
  
module_init(fh_init);  
module_exit(fh_exit);
```