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 _____ И.С.Климов
(Группа) (Подпись, дата) (И.О.Фамилия)

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

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**
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
1.4 Анализ способов перехвата функций в ядре 10
1.4.1 Модификация таблицы системных вызов 10
1.4.2 Использование сплайсинга 10
1.4.3 Использование ftrace 11
Вывод 13
2 Конструкторская часть 14
2.1 Структура программного обеспечения 20
2.2 Алгоритм перехвата функции 15
3 Технологическая часть 21
3.1 Выбор языка и среды программирования 21
3.2 21
4 Исследовательская часть 23
ЗАКЛЮЧЕНИЕ **24**
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **25**

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 деконструктору.

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 курсовой проект, необходимо разработать и отладить программное обеспечение, собирающее статистику выделения памяти slab для конкретного процесса и записывающее его в системный журнал. Для решения этой задачи необходимо:

1) изучить структуры и функции для работы со slab;
2) изучить механизмы перехвата функций в ядре.

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

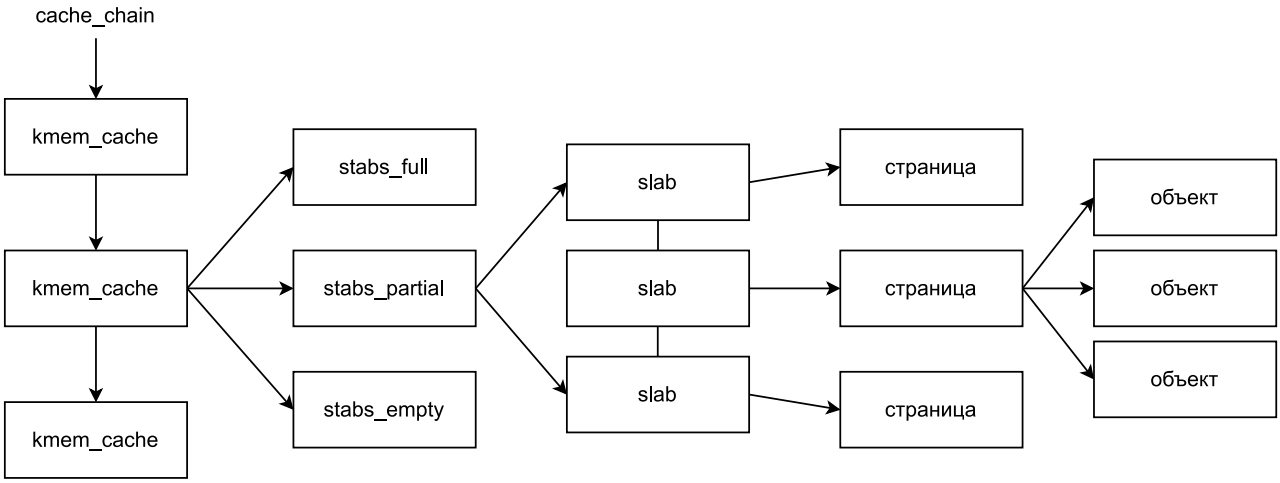

Рисунок 1.1 – Главные структуры распределителя slab

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

1) stabs_full – stab’ы, которые распределены полностью;
2) stabs_partial – slab’ы, которые распределены частично;
3) stabs_empty – stab’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 stabs_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. Функия 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_cach* 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(
        void* objp
);
```

В kmalloc необходимые для распределения аргументы – только размер объектов и набор флагов. Но kmalloc и kfree используют кэш slab точно так же, как и определенные ранее функции. Вместо того, чтобы вызывать определенный кэш slab, из которого выделяется объект, функция kmalloc повторяет через доступные кэши поиск того, который соответствует запрошенному размеру. Когда он найден, объект выделяется (при помощи kmem_cache_alloc). Чтобы освободить при помощи kfree, 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 [1].

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 kmalloc и kfree и оценивать количество выделенной памяти для исследуемого процесса.

1.4 Анализ способов перехвата функций в ядре

1.4.1 Модификация таблицы системных вызов

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 специальлной 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 дерегистрация 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**
Модификация	–	+	+	–
Сплайсинг	+	+	–	–
Ftrace	+	+	+	+

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.

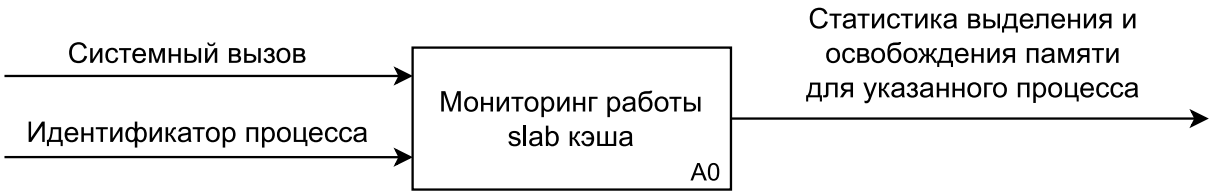

Рисунок 2.1 – IDEF0-диаграмма нулевого уровня

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

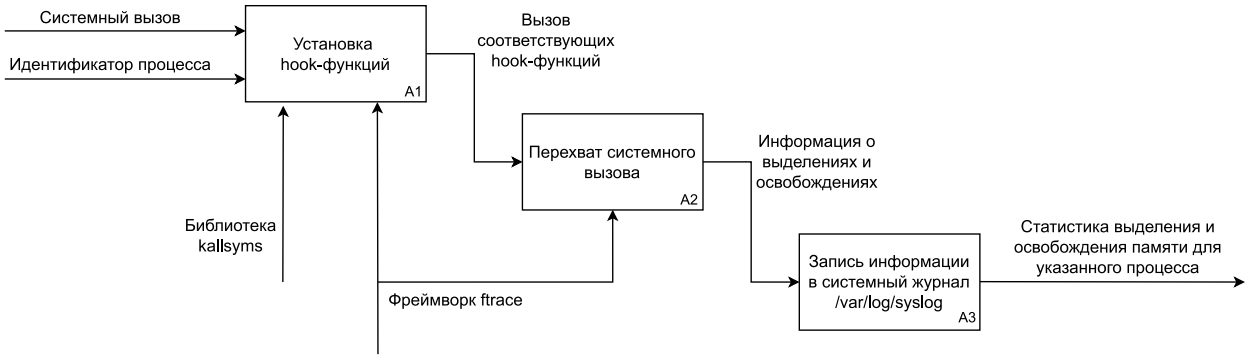

Рисунок 2.2 – IDEF0-диаграмма первого уровня

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.

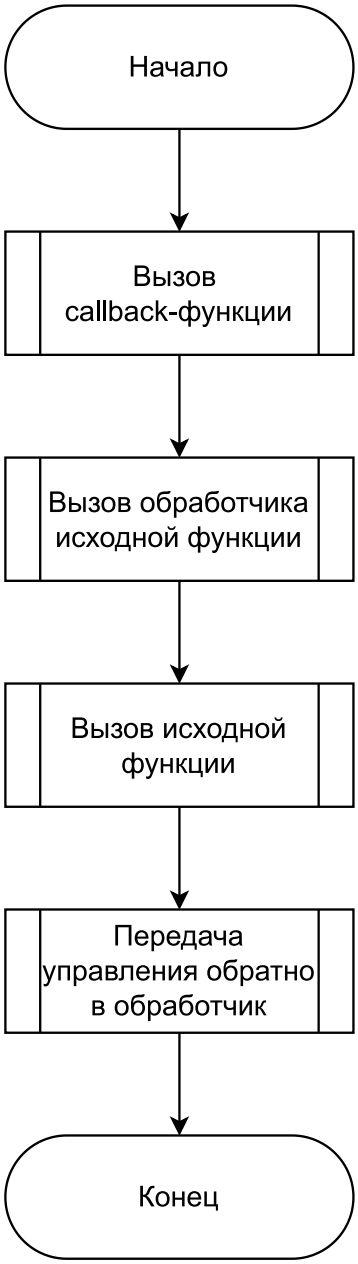

Рисунок 2.3 – Алгоритм перехвата функции

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.

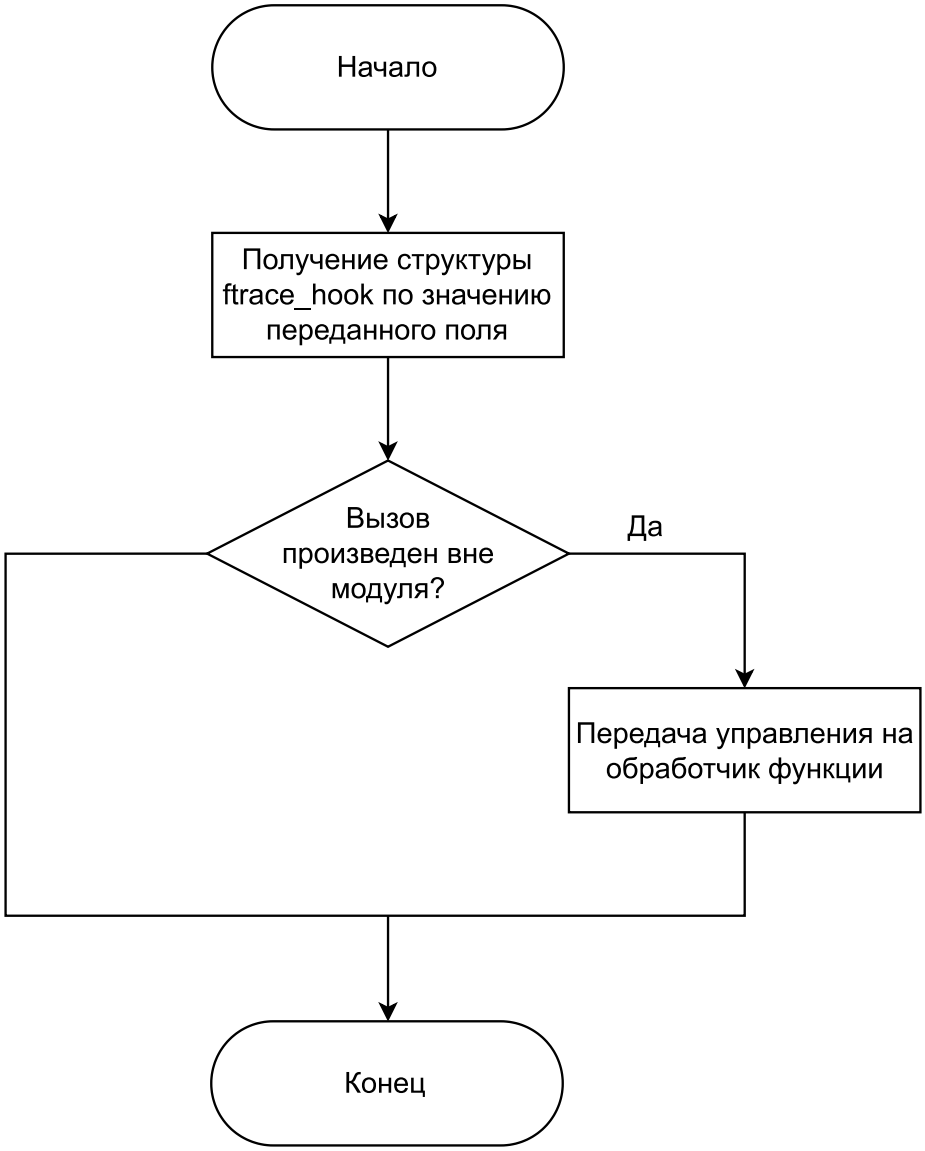

Рисунок 2.4 – Алгоритм защиты от повторного вызова функции

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 b 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.

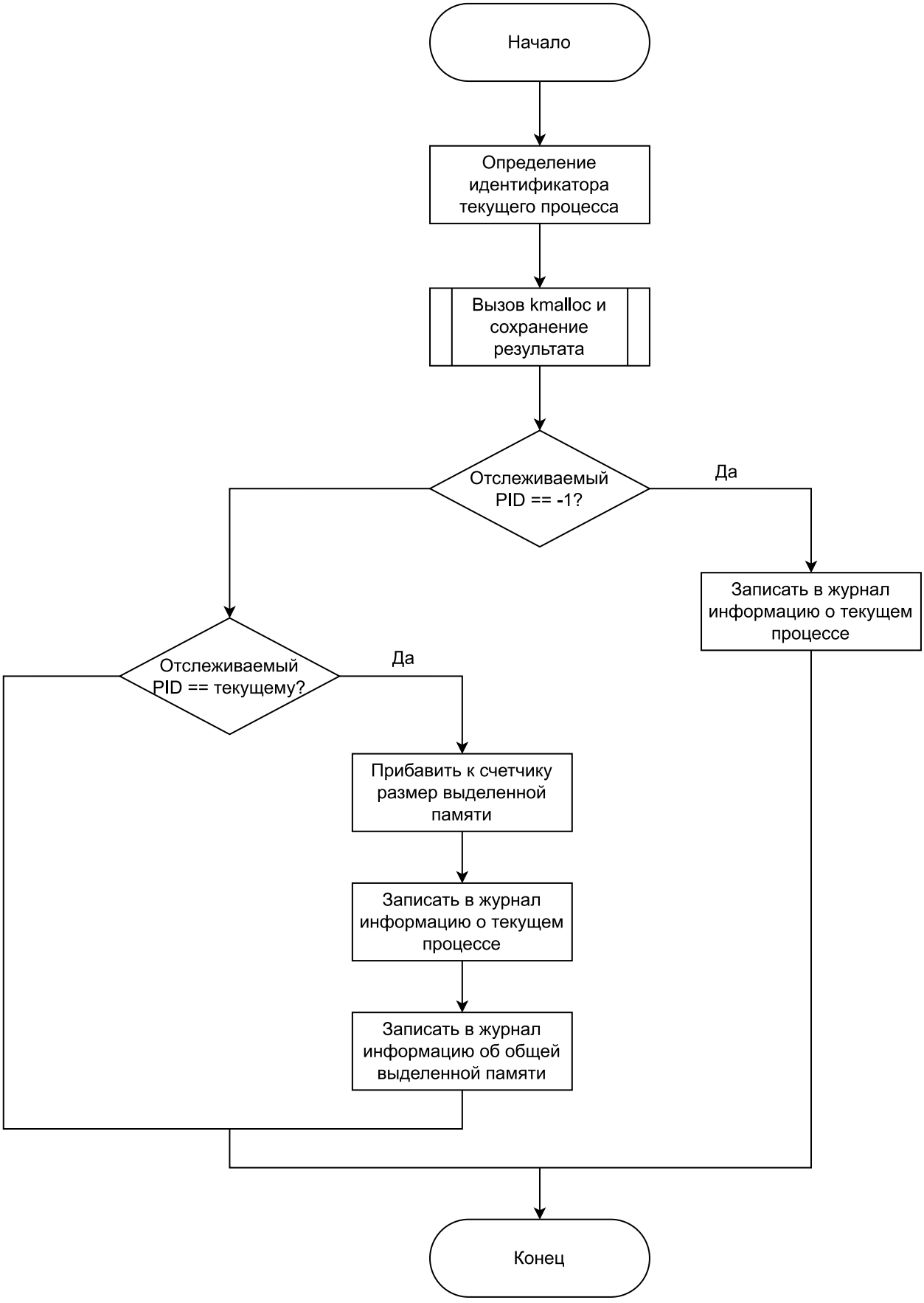

Рисунок 2.5 – Алгоритма обработчика kmalloc

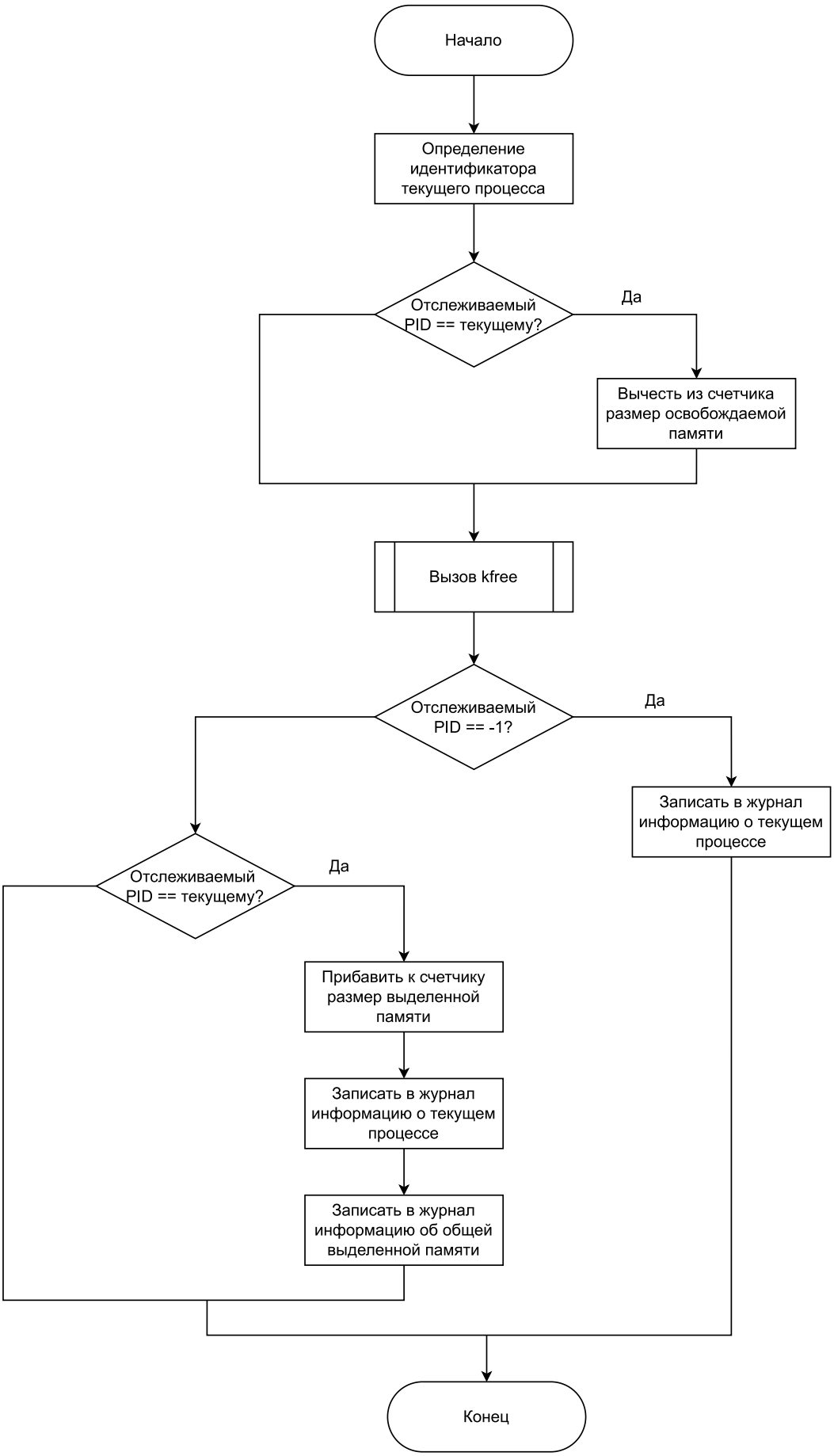

Рисунок 2.6 – Алгоритм обработчика kfree

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.

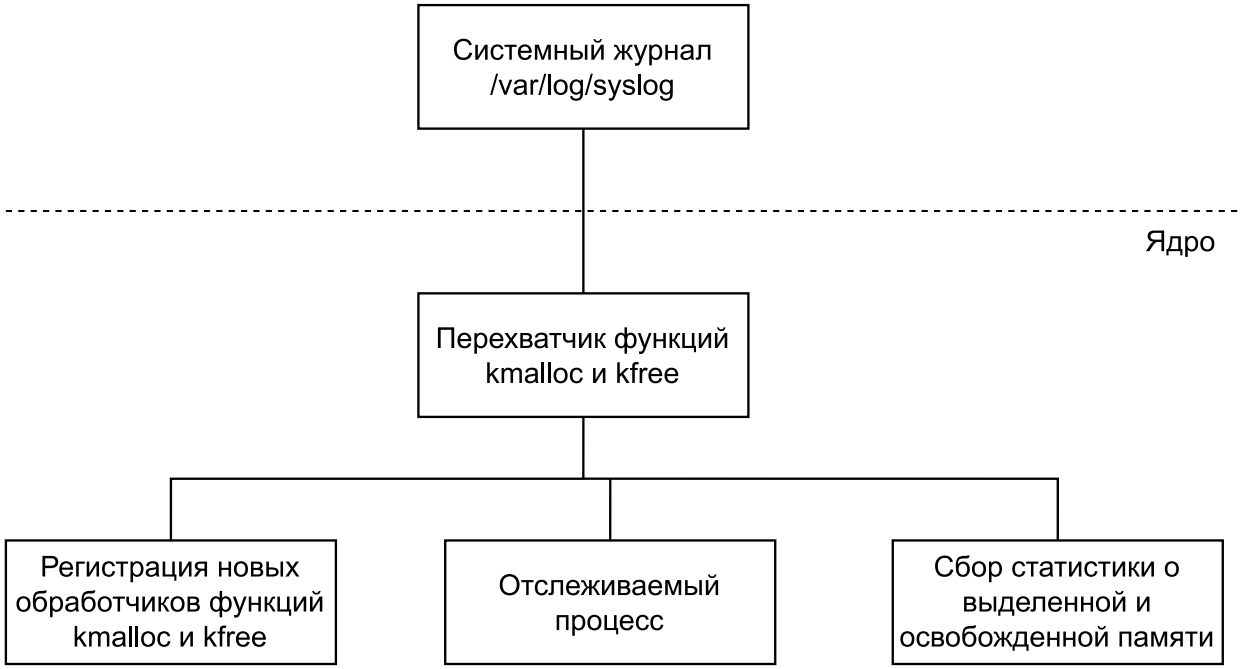

Рисунок 2.7 – Структурная схема ПО

3 Технологическая часть

3.1 Выбор языка и среды программирования

Наиболее оптимальным выбором языка программирования для написания загружаемого модуля является язык C. 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),
};
```

4 Исследовательская часть

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1.