



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 использования кэша slab*

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 .....	6
1.1 Постановка задачи .....	6
1.2 Описание принципов работы кэша slab.....	6
1.3 Описание и анализ API для работы со slab .....	7
1.4 Анализ способов перехвата функций в ядре.....	11
1.4.1 Модификация таблицы системных вызов.....	11
1.4.2 Использование сплайсинга .....	11
1.4.3 Использование ftrace .....	12
Вывод .....	13
2 Конструкторская часть .....	14
2.1 Структура программного обеспечения.....	14
2.2 Алгоритм перехвата функции.....	14
3 Технологическая часть .....	16
3.1 Выбор языка и среды программирования .....	16
3.2.....	16
4 Исследовательская часть.....	17
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>18</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....</b>	<b>19</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. В качестве статистических данных выступают:

- 1) идентификатор процесса, исполнение которого привело к вызову функции ядра;

- 2) значения аргументов вызываемой функции;
- 3) значение, возвращаемое вызываемой функцией;
- 4) суммарное количество памяти, выделенной процессу с момента загрузки данного модуля.

# 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

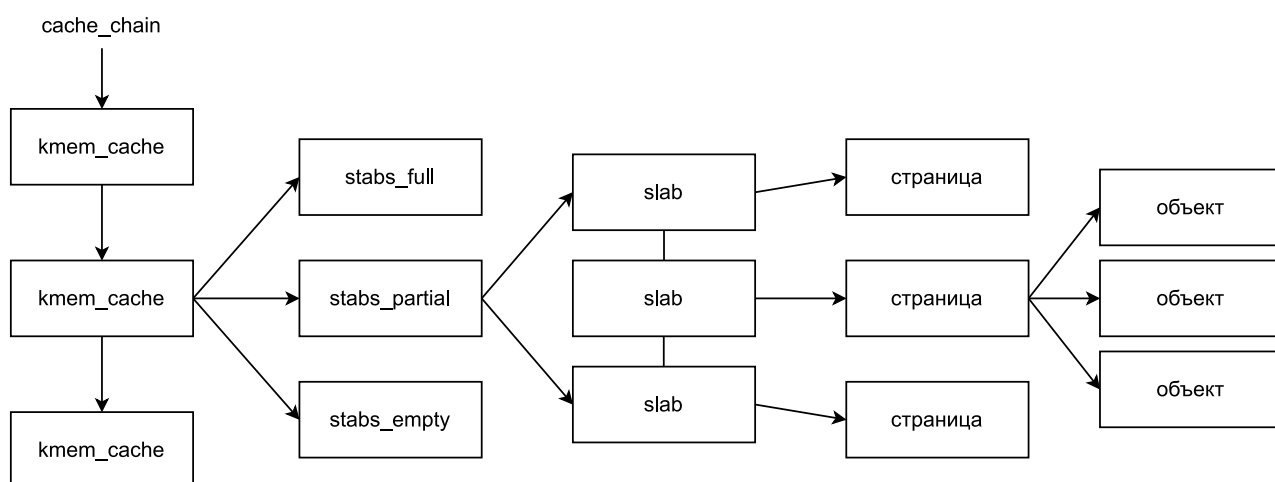


Рисунок 1.1 – Главные структуры распределителя slab

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

- 1) 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 поло, 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(  
    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 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.

## Вывод

Таким образом, `ftrace` позволяет получить управление во время входа в любую функцию ядра, и выполнить там собственный код, собирающий диагностику использования статистики использования данной функции. Использование данной технологии является целесообразным для решения поставленной задачи.

## **2 Конструкторская часть**

### **2.1 Структура программного обеспечения**

В соответствии с проведенным анализом в состав программного обеспечения будет входить модуль ядра ОС Linux, отслеживающий выход функций `kmalloc()` и `kfree()` для заданного процесса, считающий статистику выделения и освобождения памяти и записывающий ее в системный журнал. Данный 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.

### **2.2 Алгоритм перехвата функции**

Алгоритм перехвата функции представлен на рисунке 2.1.

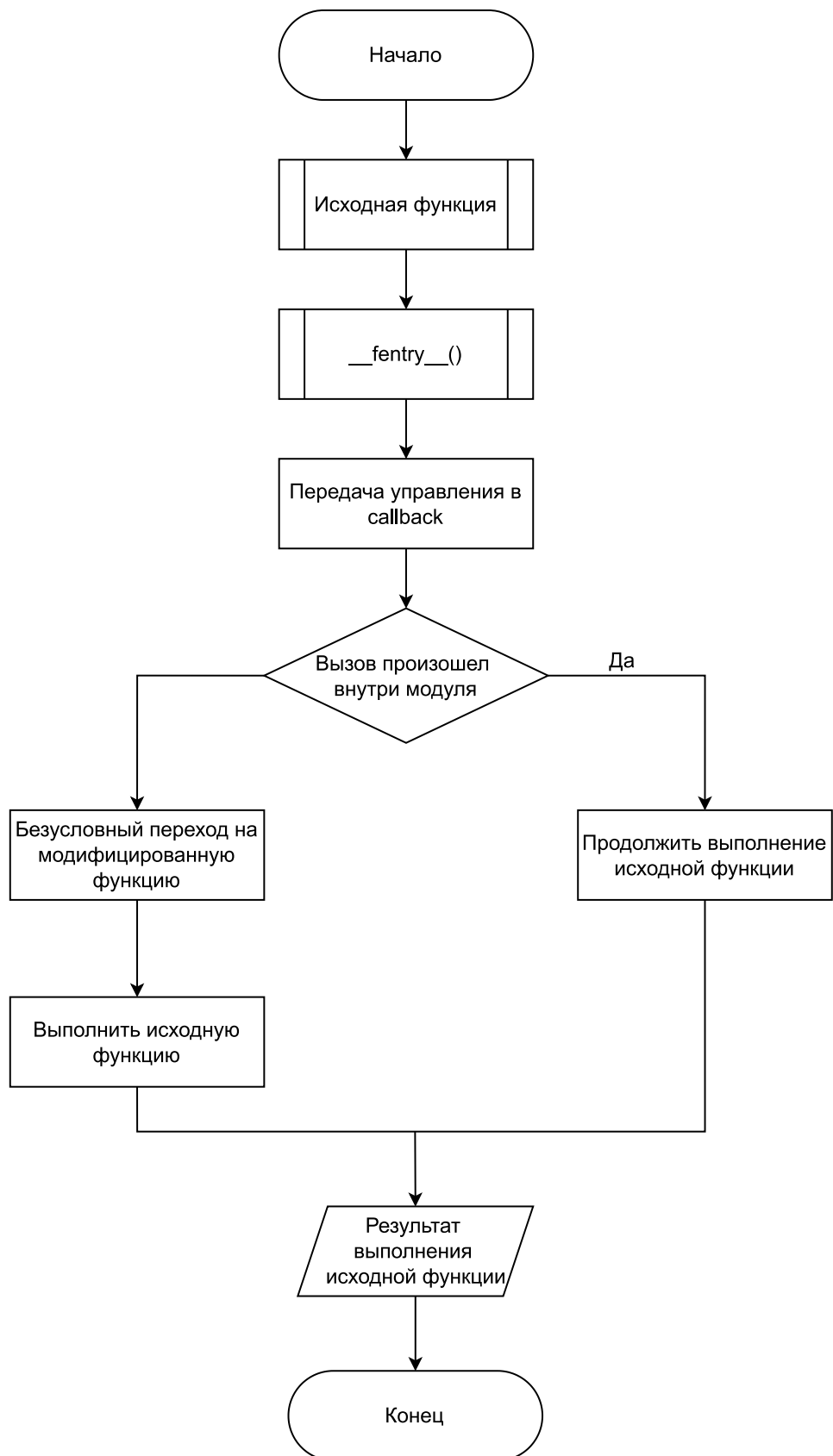


Рисунок 2.1 – Схема алгоритма перехвата функции

### **3 Технологическая часть**

#### **3.1 Выбор языка и среды программирования**

Наиболее оптимальным выбором языка программирования для написания загружаемого модуля – язык С. Для компиляции модуля используется компилятор gcc, для сборки – утилита make.

#### **3.2**

## **4 Исследовательская часть**



## **ЗАКЛЮЧЕНИЕ**

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1.