Содержание

ВВЕДЕНИЕ **4**

1 Аналитическая часть **5**

1.1 Постановка задачи 5

1.2 Анализ распределителей памяти SLAB и SLUB 5

1.3 Анализ API для работы с распределителями SLAB и SLUB 6

1.4 Механизмы перехвата функций 15

1.4.1 kprobes 15

1.4.2 ftrace 16

1.5 Виртуальная файловая система proc 18

1.6 Взаимодействие процесса с загружаемым модулем ядра 20

2 Конструкторская часть **21**

2.1 Последовательность преобразований 21

2.2 Алгоритм загрузки и выгрузки загружаемого модуля ядра 22

2.3 Алгоритм перехвата функции ядра 22

2.4 Алгоритмы чтения и записи для файла в виртуальной файловой системе proc . . 23

2.5 Алгоритмы подменяемых функций 25

3 Технологическая часть **29**

3.1 Выбор языка и среды программирования 29

3.2 Описание структур 29

3.3 Реализация загружаемого модуля ядра 30

4 Исследовательская часть **39**

4.1 Вывод 41

ЗАКЛЮЧЕНИЕ **43**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **44**

Приложение А **45**

Приложение Б **67**

ВВЕДЕНИЕ

Распределитель памяти SLAB основан на алгоритме, предложенном Джеффом Бонвиком для операционной системы SunOS [1]. Распределитель Бонвика строится вокруг объекта кэширования [1]. Внутри ядра значительное количество памяти выделяется на ограниченный набор объектов, например, дескрипторы файлов и другие общие структурные элементы. Бонвик основывался на том, что количество времени, необходимое для инициализации регулярного объекта в ядре, превышает количество времени, необходимое для его выделения и освобождения [1]. 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 [1]. Например, если память выделена для mutex, функцию инициализации mutex 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 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 кэшэй slab, но не позволяет отследить использование памяти конкретным процессом.

Целью работы является разработка загружаемого модуля ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux.

Задачи работы:

— анализ и выбор методов и средств реализации загружаемого модуля ядра;

— разработка структур и алгоритмов, необходимых для работы загружаемого модуля ядра;

— анализ результатов работы разработанного загружаемого модуля ядра.

1 Аналитическая часть

1.1 Постановка задачи

В рамках выполнения курсовой работы необходимо разработать загружаемый модуль ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux. Для реализации поставленной задачи необходимо:

— провести анализ распределителей памяти SLAB и SLUB;

— провести анализ и выбор механизмов перехвата функций;

— провести анализ методов передачи информации из пространства пользователя в пространство ядра и наоборот;

— разработать структуры и алгоритмы загржаемого модуля ядра;

— реализовать загружаемый модуль ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux;

— провести анализ работы реализованного загружаемого модуля ядра.

К разрабатываемому модулю ядра предъявляются следующие требования:

— обеспечение передачи данных из пространства пользователя в пространство ядра и наоборот;

— возможность взаимодействия процессов из пространства пользователя с разработанным загружаемым модулем ядра;

— обеспечения сбора статистики использования SLAB-кэша для одного и более процессов;

— получение и хранение информации о SLAB-кэшах — PID процесса, имя кэша, число выделенных объектов, размер объекта, число объектов в одном slab, число страниц в одном slab, общее число выделенных страниц.

1.2 Анализ распределителей памяти SLAB и SLUB

Для многих структур, используемых в ядре, время, необходимое для инициализации объекта, превышает время, затрачиваемое на выделение для него памяти [1]. Для решения данной проблемы был разработан распределитель SLAB, основная идея которого заключается в хранении часто используемых объектов в инициализированном состоянии, доступном для использования ядром [1].

Взаимосвязь между составляющими распределителя SLAB представлена на рисунке 1.1.

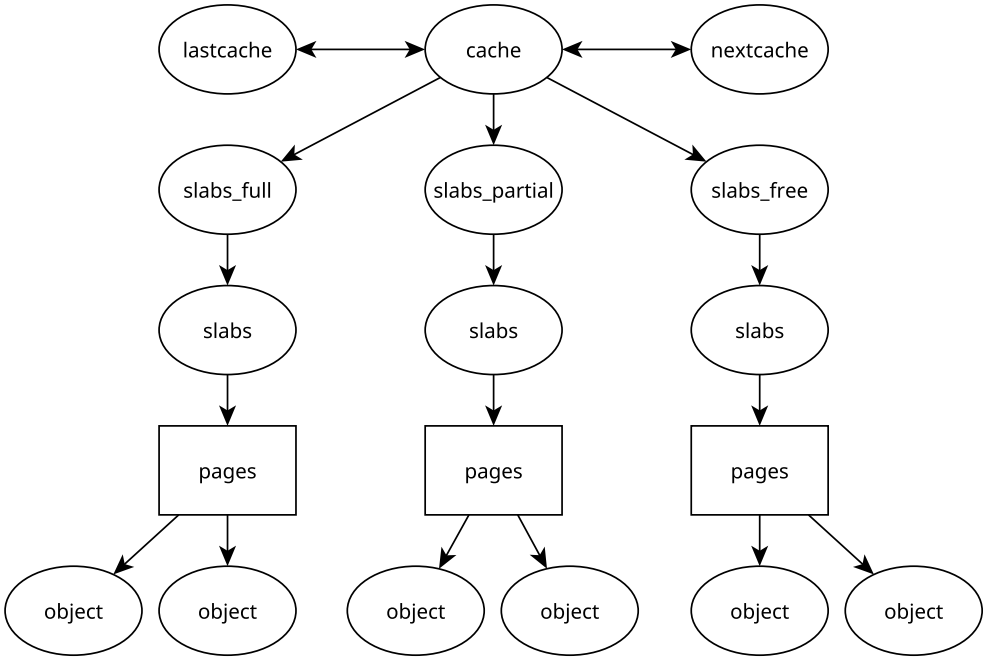

Рисунок 1.1 — Схема распределителя SLAB

Распределитель SLAB состоит из переменного количества кэшей, которые содержатся в двусвязном циклическом списке [1]. Каждый кэш хранит в памяти блоки смежных страниц, которые разделены на небольшие фрагменты для хранения структур данных и объектов, которыми он управляет [1]. Существуют три вида фрагментов SLAB [1]:

— slabs_full (полностью распределенные фрагменты);

— slabs_partial (частично распределенные фрагменты);

— slabs_empty (пустые фрагменты, не выделенные под объекты).

Поскольку объекты выделяются и освобождаются, отдельные фрагменты SLAB могут перемещаться между соответствующими списками [1]. Когда все объекты фрагмента SLAB израсходованы, система переносит их из списка slabs_partial в список slabs_full [1]. Когда фрагмент SLAB полон, и объект освобождается, он перемещается из списка slabs_full в список slabs_partial [1]. Когда освобождаются все объекты фрагмента, они перемещаются в список slabs_empty [1].

Начиная с версии 2.6.23 ядра Linux используется SLUB — усовершенствованный распределитель SLAB [2]. В нем используется базовая модель SLAB, однако исправлены некоторые недостатки, особенно для систем с большим количеством процессоров [2].

1.3 Анализ API для работы с распределителями SLAB и SLUB

Основной структурой для работой с распределителями SLAB и SLUB ялвляется struct kmem_cache, содержащий информацию о кэше. Объявление структуры struct kmem_cache для распределителя SLAB для ядра Linux версии 6.5.13 представлено в листинге 1.1.

Листинг 1.1 — Объявление структуры struct kmem_cache для распределителя SLAB (версия ядра Linux — 6.5.13)

```
struct kmem_cache{
  struct array_cache __percpu *cpu_cache;

  /* 1) Cache tunables. Protected by slab_mutex */
  unsigned int batchcount;
  unsigned int limit;
  unsigned int shared;

  unsigned int size;
  struct reciprocal_value reciprocal_buffer_size;
  /* 2) touched by every alloc & free from the backend */

  slab_flags_t flags;   /* constant flags */
  unsigned int num;   /* # of objs per slab */

  /* 3) cache_grow/shrink */
  /* order of pgs per slab (2^n) */
  unsigned int gfporder;

  /* force GFP flags, e.g. GFP_DMA */
  gfp_t allocflags;

  size_t colour;      /* cache colouring range */
  unsigned int colour_off;  /* colour offset */
  unsigned int freelist_size;

  /* constructor func */
  void ( *ctor)(void *obj);

  /* 4) cache creation/removal */
  const char *name;
  struct list_head list;
  int refcount;
  int object_size;
  int align;

  /* 5) statistics */
  #ifdef CONFIG_DEBUG_SLAB
  unsigned long num_active;
```

```
  unsigned long num_allocations;
  unsigned long high_mark;
  unsigned long grown;
  unsigned long reaped;
  unsigned long errors;
  unsigned long max_freeable;
  unsigned long node_allocs;
  unsigned long node_frees;
  unsigned long node_overflow;
  atomic_t allochit;
  atomic_t allocmiss;
  atomic_t freehit;
  atomic_t freemiss;

  /*
  * If debugging is enabled, then the allocator can add additional
  * fields and/or padding to every object. 'size' contains the total
  * object size including these internal fields, while 'obj_offset'
  * and 'object_size' contain the offset to the user object and its
  * size.
  */
  int obj_offset;
  #endif /* CONFIG_DEBUG_SLAB */

  #ifdef CONFIG_KASAN_GENERIC
  struct kasan_cache kasan_info;
  #endif

  #ifdef CONFIG_SLAB_FREELIST_RANDOM
  unsigned int *random_seq;
  #endif

  #ifdef CONFIG_HARDENED_USERCOPY
  unsigned int useroffset;  /* Usercopy region offset */
  unsigned int usersize;    /* Usercopy region size */
  #endif

  struct kmem_cache_node *node[MAX_NUMNODES];
};
```

Объявление структуры struct kmem_cache для распределителя SLAB для ядра Linux

версии 6.5.13 представлено в листинге 1.2.

Листинг 1.2 — Объявление структуры struct kmem_cache для распределителя SLUB (версия ядра Linux — 6.5.13)

```
struct kmem_cache{
  #ifndef CONFIG_SLUB_TINY
  struct kmem_cache_cpu __percpu *cpu_slab;
  #endif
  /* Used for retrieving partial slabs, etc. */
  slab_flags_t flags;
  unsigned long min_partial;
  unsigned int size;  /* The size of an object including metadata */
  unsigned int object_size;/* The size of an object without metadata */
  struct reciprocal_value reciprocal_size;
  unsigned int offset;  /* Free pointer offset */
  #ifdef CONFIG_SLUB_CPU_PARTIAL
  /* Number of per cpu partial objects to keep around */
  unsigned int cpu_partial;
  /* Number of per cpu partial slabs to keep around */
  unsigned int cpu_partial_slabs;
  #endif
  struct kmem_cache_order_objects oo;

  /* Allocation and freeing of slabs */
  struct kmem_cache_order_objects min;
  gfp_t allocflags; /* gfp flags to use on each alloc */
  int refcount;   /* Refcount for slab cache destroy */
  void ( *ctor)(void *);
  unsigned int inuse;   /* Offset to metadata */
  unsigned int align;   /* Alignment */
  unsigned int red_left_pad;  /* Left redzone padding size */
  const char *name; /* Name (only for display!) */
  struct list_head list;  /* List of slab caches */
  #ifdef CONFIG_SYSFS
  struct kobject kobj;  /* For sysfs */
  #endif
  #ifdef CONFIG_SLAB_FREELIST_HARDENED
  unsigned long random;
  #endif

  #ifdef CONFIG_NUMA
  /*
```

```
  * Defragmentation by allocating from a remote node.
  */
  unsigned int remote_node_defrag_ratio;
  #endif

  #ifdef CONFIG_SLAB_FREELIST_RANDOM
  unsigned int *random_seq;
  #endif

  #ifdef CONFIG_KASAN_GENERIC
  struct kasan_cache kasan_info;
  #endif

  #ifdef CONFIG_HARDENED_USERCOPY
  unsigned int useroffset;  /* Usercopy region offset */
  unsigned int usersize;    /* Usercopy region size */
  #endif

  struct kmem_cache_node *node[MAX_NUMNODES];
};
```

Для создания нового SLAB-кэша применяется функция kmem_cache_create [1], прототип которой представлен в листинге 1.3.

Листинг 1.3 — Прототип функции kmem_cache_create (версия ядра Linux — 6.5.13)

```
struct kmem_cache *kmem_cache_create(
  const char *name,        /* имя кэша */
  unsigned int size,       /* размер выделяемых в кэше объектов */
  unsigned int align,      /* выравнивание объектов */
  slab_flags_t flags,      /* флаги SLAB */
  void ( *ctor)(void *)    /* конструктор выделяемых в кэше объектов */
);
```

Существуют следующие флаги SLAB [3]:

— SLAB_POISON — запись в SLAB шаблонного значения 0x5a5a5a5a (используется для получения ссылок на неинициализированную память);

— SLAB_RED_ZONE — вставка «красных зон» в выделенные участки памяти для отслеживания переполнения;

— SLAB_HWCACHE_ALIGN — выделение объектов в кэше по аппаратной линии кэширования.

Для уничтожения SLAB-кэша применяется системный вызов kmem_cache_destroy [1]. Прототип функции kmem_cache_destroy представлен в листинге 1.4. Диаграмма вызовов для функции kmem_cache_destroy представлена на рисунке 1.2.

Листинг 1.4 — Прототип функции kmem_cache_create (версия ядра Linux — 6.5.13)

```
void kmem_cache_destroy(
  struct kmem_cache *s  /* указатель на SLABкэш- */
);
```

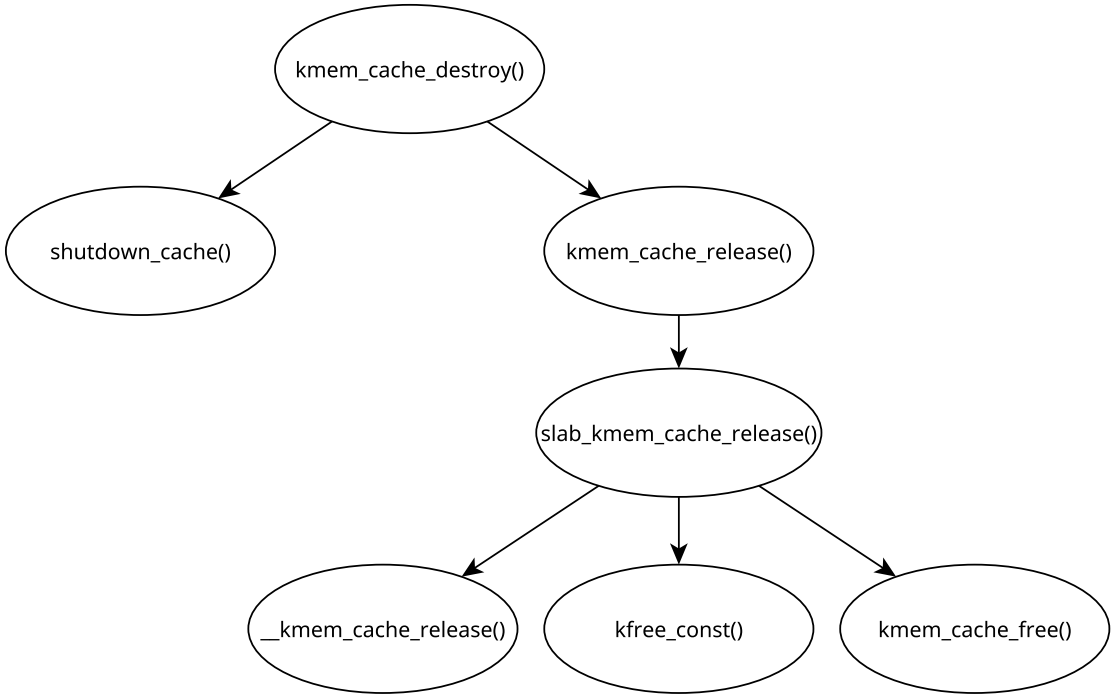

Рисунок 1.2 — Диаграмма вызовов для функции kmem_cache_destroy (версия ядра Linux — 6.5.13)

Для выделения памяти из SLAB-кэша применяются системные вызовы kmem_cache_alloc и kmalloc [1]. Прототипы функций kmem_cache_alloc и kmalloc представлены в листингах 1.5 и 1.6 соответственно.

Листинг 1.5 — Прототип функции kmem_cache_alloc (версия ядра Linux — 6.5.13)

```
void *kmem_cache_alloc(
  struct kmem_cache *cachep,  /* указатель на SLABкэш- */
  gfp_t flags                 /* флаги GFP, определяющие поведение
                                 при выделении памяти */
);
```

Листинг 1.6 — Прототип функции kmalloc (версия ядра Linux — 6.5.13)

```
void *kmalloc(
  size_t size,  /* размер выделяемого объекта */
  gfp_t flags   /* флаги GFP, определяющие поведение
                   при выделении памяти */
);
```

Существуют следующие GFP-флаги [4]:

— GFP_ATOMIC — процесс не может уснуть во время выделения памяти;

— GFP_NOIO — запрет на операции ввода/вывода во время выделения памяти;

— GFP_NOHIGHIO — используется системным вызовом alloc_bounce_page() во время создания буфера отказов для ввода-вывода в верхней памяти;

— GFP_NOFS — используется только буферным кэшем и файловыми системами для избежания рекурсии;

— GFP_KERNEL — выделение памяти от имени процесса в пространстве ядра;

— GFP_USER — выделение памяти от имени процесса в пространстве пользователя;

— GFP_HIGHUSER — выделение страниц из верхней памяти от имени пользователя.

На рисунках 1.3 и 1.4 представлены диаграммы вызовов для функций kmem_cache_alloc и kmalloc соответственно.

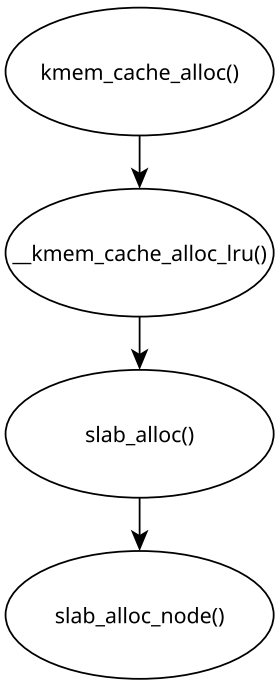

Рисунок 1.3 — Диаграмма вызовов для функции kmem_cache_alloc (версия ядра Linux — 6.5.13)

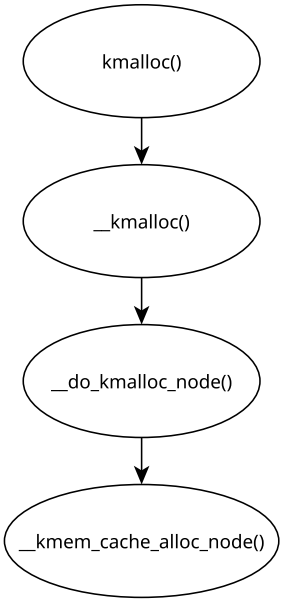

Рисунок 1.4 — Диаграмма вызовов для функции kmalloc (версия ядра Linux — 6.5.13)

В отличие от kmem_cache_alloc, функция kmalloc не принимает в качестве параметра указатель на объект структуры struct kmem_cache. Системный вызов kmalloc осуществляет поиск кэша, который соответствует указанному размеру, и передает в качестве параметра указатель на него функции __kmem_cache_alloc_node для последующего выделения памяти.

Для освобождения выделенной из SLAB-кэша памяти используются системные вызовы kmem_cache_free и kfree [1]. Прототипы функций kmem_cache_free и kfree представлены в листингах 1.7 и 1.8 соответственно.

Листинг 1.7 — Прототип функции kmem_cache_free (версия ядра Linux — 6.5.13)

```
void kmem_cache_free(
  struct kmem_cache *s,  /* указатель на SLABкэш- */
  void *objp             /* указатель на освобождаемый объект */
);
```

Листинг 1.8 — Прототип функции kfree (версия ядра Linux — 6.5.13)

```
void kfree(
  const void *objp  /* указатель на освобождаемый объект */
);
```

На рисунках 1.5 и 1.6 представлены диаграммы вызовов для функций kmem_cache_free и kfree соответственно.

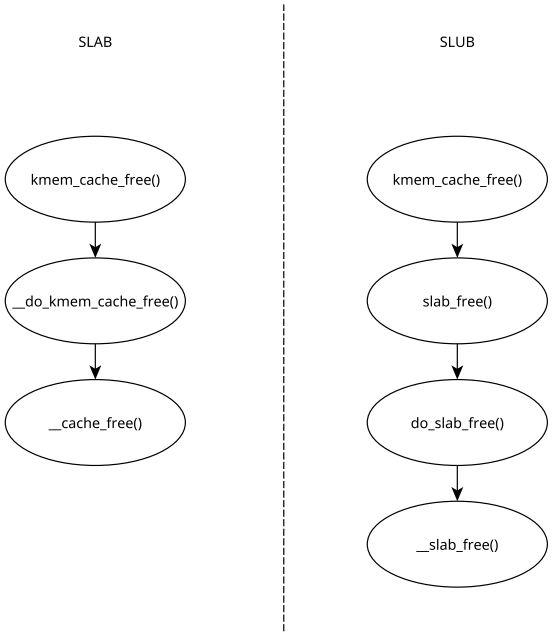

Рисунок 1.5 — Диаграмма вызовов для функции kmem_cache_free (версия ядра Linux — 6.5.13)

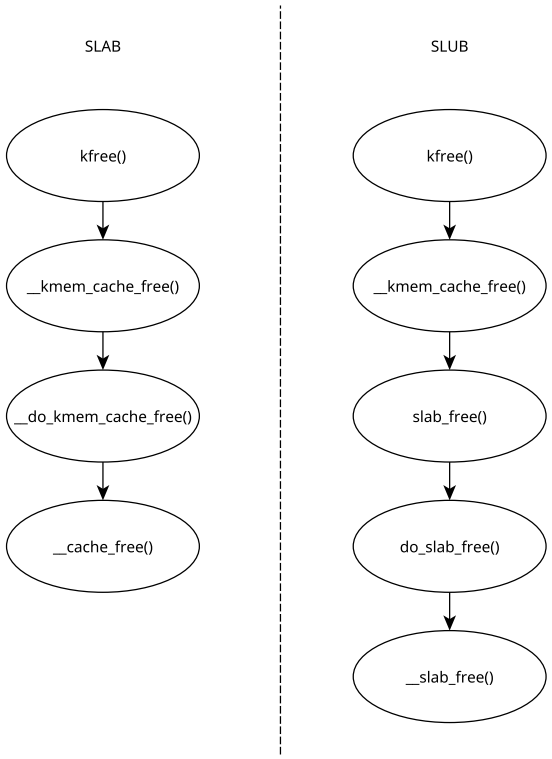

Рисунок 1.6 — Диаграмма вызовов для функции kfree (версия ядра Linux — 6.5.13)

Системный вызов kfree осуществляет поиск кэша, из которого был выделен объект, и передает в качестве параметра указатель на него функции __kmem_cache_free для последующего освобождения объекта.

Таким образом, для сбора информации об использовании SLAB-кэша необходимо оосуществить перехват следующих функций:

— __kmem_cache_alloc_node,

— __kmem_cache_free,

— kmem_cache_alloc,

— kmem_cache_free,

— kmem_cache_destroy.

1.4 Механизмы перехвата функций

Идея перехвата функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции так, чтобы при вызове перехватываемой функции управление передавалось подменяемой функции. Данная функция выполняется вместо системной функции, производя необходимые действия до и после вызова оригинальной функции.

В данной работе перехват функций необходим для получения статистики использования SLAB-кэшей и и информации о них, поскольку структура struct kmem_cache не содержит информацию о процессах, использующих данный кэш.

Существуют следующие наиболее известные подходы перехвата функций: kprobes [5] и ftrace [6].

1.4.1 kprobes

kprobes представляет собой специальный интерфейс, предназначенный для отладки и трассировки ядра [5]. Данный интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции [5]. Обработчики получают доступ к регистрам и могут изменять их значение, что позволяет использовать kprobes как в целях мониторинга, так и для влияния на дальнейшую работу ядра [5].

Интерфейс kprobes имеет следующие особенности [5]:

— перехват любой инструкции в ядре реализуется с помощью точек останова, внедряемых в исполняемый код ядра;

— относительно большие накладные расходы (для расстановки и обработки точек останова необходимо большое количество процессорного времени);

— техническая сложность реализации (в частности, чтобы получить аргументы функции или значения её локальных переменных, нужно извлекать их из регистров или стека).

1.4.2 ftrace

ftrace — фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора pg и mfentry [7]. Данные функции вставляют в начало каждой функции вызов специальной трассировочной функции mcount() или __fentry__() [7]. В пользовательских программах данная возможность компилятора используется профилировщиками, целью отслеживания всех вызываемых функций [7]. В ядре эти функции используются исключительно для реализации рассматриваемого фреймворка [7].

Для большинства современных архитектур процессора доступна оптимизация — динамический frace [7]. Ядро знает расположение всех вызовов функций mcount() или __fentry__() и на ранних этапах загрузки ядра подменяет их машинный код на специальную машинную инструкцию NOP [8], которая ничего не делает [7]. При включении трассировки, в нужные функции необходимые вызовы добавляются обратно. Если ftrace не используется, его влияние на производительность системы минимально.

Фреймворк ftrace имеет следующие особенности [7]:

— возможность перехвата любой функции;

— совместимость перехвата функции с трассировкой;

— фреймворк зависит от конфигурации ядра, при этом в популярных конфигурациях ядра установлены все необходимые флаги для работы.

Для регистрации callback-функции необходима структура struct ftrace_ops [6]. Эта структура используется, чтобы сообщить ftrace, какая callback-функция должна вызываться, а также какая защита будет выполнятся обратным вызовом и не потребует обработки ftrace [6]. Объявление структуры struct ftrace_ops представлено в листинге 1.9.

Листинг 1.9 — Объявление структуры struct ftrace_ops (версия ядра Linux — 6.5.13)

```
struct ftrace_ops{
  ftrace_func_t func;
  struct ftrace_ops __rcu *next;
  unsigned long flags;
  void *private;
  ftrace_func_t saved_func;
  #ifdef CONFIG_DYNAMIC_FTRACE
  struct ftrace_ops_hash local_hash;
  struct ftrace_ops_hash *func_hash;
  struct ftrace_ops_hash old_hash;
  unsigned long trampoline;
  unsigned long trampoline_size;
  struct list_head list;
  ftrace_ops_func_t ops_func;
  #ifdef CONFIG_DYNAMIC_FTRACE_WITH_DIRECT_CALLS
  unsigned long direct_call;
  #endif
  #endif
};
```

Для регистрации callback-функции необходимо определить поле func объекта структуры struct ftrace_ops и вызвать функцию register_ftrace_function [6]. Для дерегистрации callback-функции необходимо применить системный вызов unregister_ftrace_function [6]. Прототипы функций register_ftrace_function и unregister_ftrace_function представлены в листингах 1.10 и 1.11 соотвественно.

Листинг 1.10 — Прототип функции register_ftrace_function (версия ядра Linux — 6.5.13)

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

Листинг 1.11 — Прототип функции unregister_ftrace_function (версия ядра Linux — 6.5.13)

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

Callback-функция должна иметь прототип, представленный в листинге 1.12.

Листинг 1.12 — Прототип callback-функции для работы с ftrace

```
void callback_func(
  unsigned long ip,          /* указатель на отслеживаемую функцию */
  unsigned long parent_ip,   /* указатель на функцию, вызвавшая
                                отслеживаемую функцию */
  struct ftrace_ops *op,
  struct pt_regs *regs       /* указатель на объект структуры,
                                позволяющего установить значения
                                в регистрах процессора после
                                выхода из callback функции,
                                если был установлен соответствующий
                                флаг */
);
```

Для перехвата функции необходимо в callback-функции изменить поле ip объекта regs структуры struct pt_regs на адрес новой функции.

1.5 Виртуальная файловая система proc

Виртуальная файловая система proc — специальный интерфейс, с помощью которого можно мгновенно получить некоторую информацию о ядре в пространстве пользователя и передать информацию в пространство ядра [8].

Директория /proc содержит, в частности, по одному подкаталогу для каждого запущенного в системе процесса [8]. Имя каждого такого подкаталога является числом, значение которого равно PID соответствующего процесса [8]. Помимо информации о каждом запущенном процессе виртуальная файловая система proc предоставляет данные о работе ядра системы [8].

Для работы с файлами в виртуальной файловой системе proc используется структура struct proc_ops, содержащая указатели на функции взаимодействия с файлом, такие как открытие, закрытие, чтение и запись. Объявление структуры struct proc_ops представлено в листинге 1.13.

Листинг 1.13 — Объявление структуры struct proc_ops (версия ядра Linux — 6.5.13)

```
struct proc_ops
{
  unsigned int proc_flags;
  int ( *proc_open)(struct inode *, struct file *);
  ssize_t ( *proc_read)(struct file *, char __user *, size_t, loff_t *)
   ;
  ssize_t ( *proc_read_iter)(struct kiocb *, struct iov_iter *);
  ssize_t ( *proc_write)(struct file *, const char __user *, size_t,
   loff_t *);
  /* mandatory unless nonseekable_open() or equivalent is used */
  loff_t  ( *proc_lseek)(struct file *, loff_t, int);
  int ( *proc_release)(struct inode *, struct file *);
  __poll_t ( *proc_poll)(struct file *, struct poll_table_struct *);
  long  ( *proc_ioctl)(struct file *, unsigned int, unsigned long);
  #ifdef CONFIG_COMPAT
  long  ( *proc_compat_ioctl)(struct file *, unsigned int, unsigned
   long);
  #endif
  int ( *proc_mmap)(struct file *, struct vm_area_struct *);
  unsigned long ( *proc_get_unmapped_area)(struct file *, unsigned long
   , unsigned long, unsigned long, unsigned long);
} __randomize_layout;
```

Для создания файла в виртуальной файловой системе proc используется системный вызов proc_create, прототип которого представлен в листинге 1.14.

Листинг 1.14 — Прототип функции proc_create (версия ядра Linux — 6.5.13)

```
struct proc_dir_entry *proc_create(
  const char *name,                 /* имя файла */
  umode_t mode,                     /* флаги, описывающие разрешения
                                       на чтение и запись */
  struct proc_dir_entry *parent,    /* указатель на объект структуры
                                       proc_dir_entry, описывающий
                                       родительскую директорию
                                       ( если parent равно NULL, то файл
                                       создаётся в директории /proc) */
  const struct proc_ops *proc_ops   /* указатель на объект структуры
                                       proc_ops, содержащий
                                       указатели на функции
                                       работы с файлом */
);
```

Функция proc_create возвращает указатель на объект структуры proc_dir_entry созданного файла при успехе или NULL при неудаче.

1.6 Взаимодействие процесса с загружаемым модулем ядра

Для взаимодействия приложений с ядром и ядра с приложениями используются функции ядра copy_to_user и copy_from_user.

Функция copy_to_user копирует данные из пространства ядра в пространство пользователя. Прототип функции copy_to_user представлен в листинге 1.15.

Листинг 1.15 — Прототип функции copy_to_user (версия ядра Linux — 6.5.13)

```
long copy_to_user(
  void __user *to,    /* адрес буфера в пространстве пользователя */
  const void *from,   /* адрес буфера в пространстве ядра */
  long n              /* количество копируемых байт */
);
```

Функция copy_to_user возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

Функция copy_from_user копирует данные из пространства пользователя в пространство ядра. Прототип функции copy_from_user представлен в листинге 1.16.

Листинг 1.16 — Прототип функции copy_to_user (версия ядра Linux — 6.5.13)

```
long copy_from_user(
  void __user *to,    /* адрес буфера в пространстве ядра */
  const void *from,   /* адрес буфера в пространстве пользователя */
  long n              /* количество копируемых байт */
);
```

Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

Вывод

Была поставлена задача по разработке загружаемого модуля ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux. Был проведен анализ распределителей SLAB и SLUB и API для работы с ними, механизмы перехвата функций kprobes и ftrace, виртуальной файловой системы proc.

Для перехвата функции был выбран фреймворк ftrace, поскольку он позволяет перехватит любую функцию по ее имени, загружается в ядро динамически и имеет задокументированный API.

2 Конструкторская часть

2.1 Последовательность преобразований

На рисунках 2.1 и 2.2 представлены IDEF0-диаграммы разрабатываемого загружаемого модуля ядра.

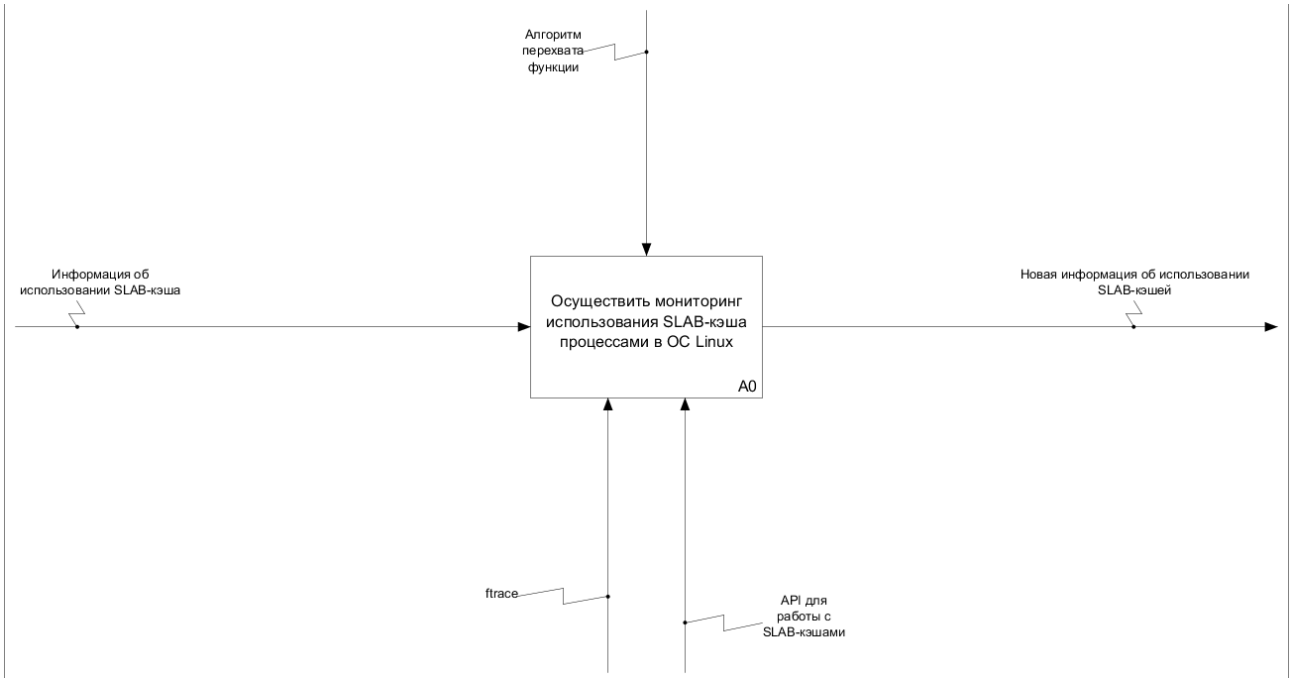

Рисунок 2.1 — IDEF0-диаграмма нулевого уровня

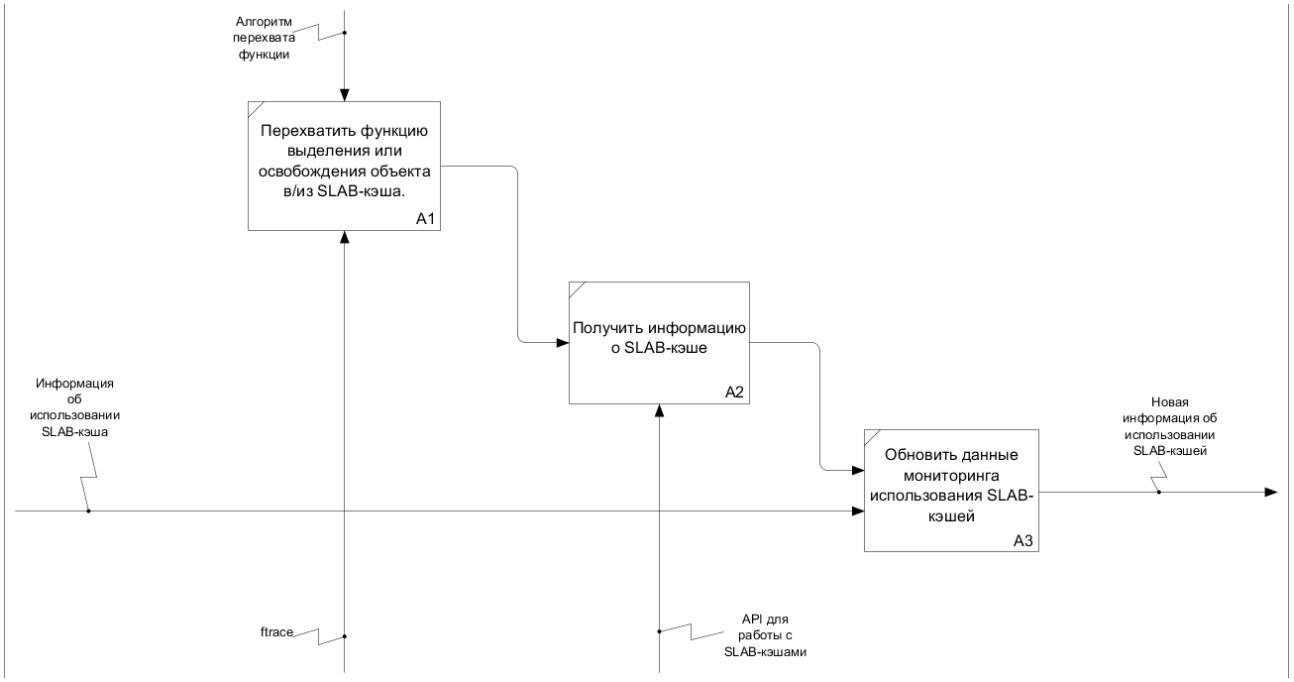

Рисунок 2.2 — IDEF0-диаграмма первого уровня

2.2 Алгоритм загрузки и выгрузки загружаемого модуля ядра

На рисунке 2.3 представлены схемы алгоритмов загрузки и выгрузки загружаемого модуля ядра.

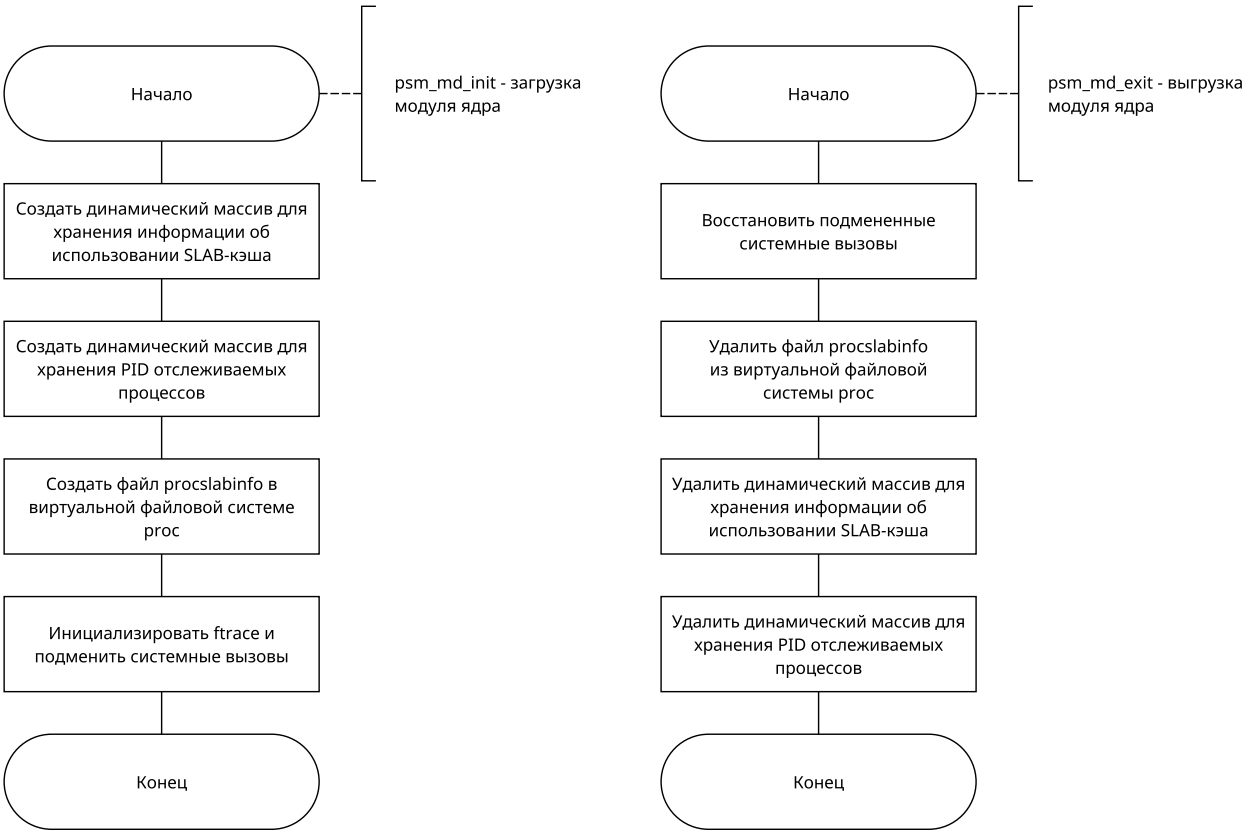

Рисунок 2.3 — Схемы алгоритмов загрузки и выгрузки загружаемого модуля ядра

2.3 Алгоритм перехвата функции ядра

На рисунке 2.4 представлена схема алгоритма перехвата функции ядра.

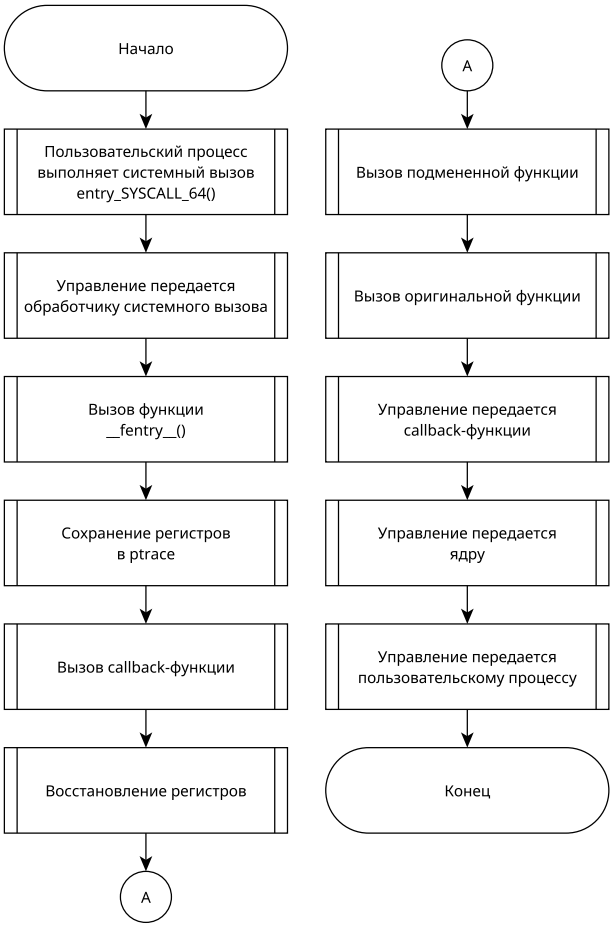

Рисунок 2.4 — Схема алгоритма перехвата функции ядра

2.4 Алгоритмы чтения и записи для файла в виртуальной файловой системе proc

На рисунках 2.5 и 2.6 представлены схемы алгоритмов чтения и записи для файла в виртуальной файловой системе proc соответственно.

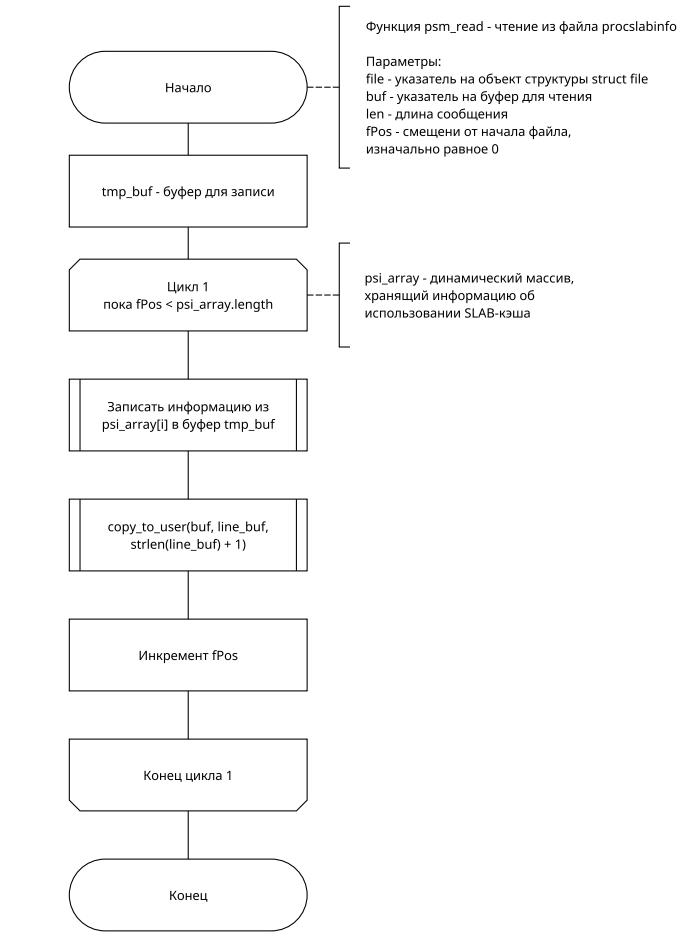

Рисунок 2.5 — Схема алгоритма чтения из файла в виртуальной файловой системе proc

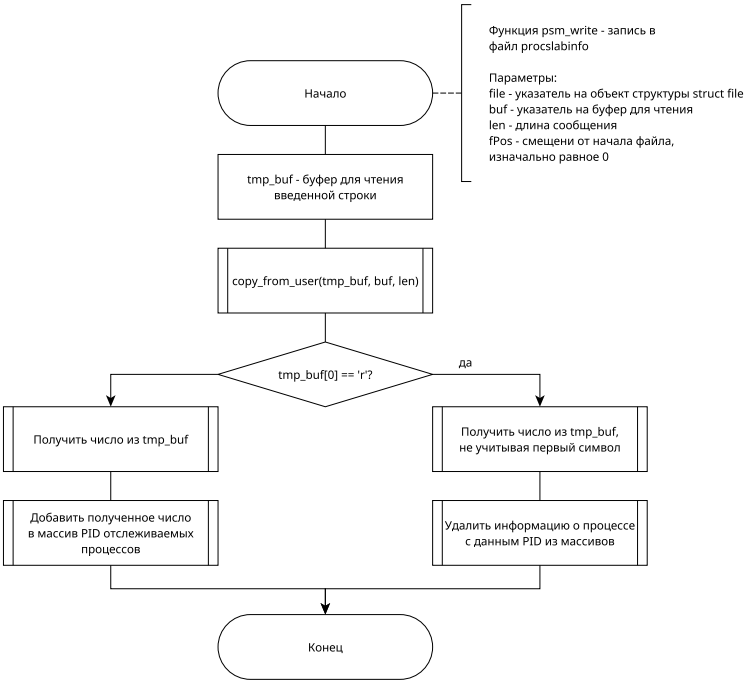

Рисунок 2.6 — Схема алгоритма записи в файл в виртуальной файловой системе proc

2.5 Алгоритмы подменяемых функций

На рисунках 2.7 – 2.11 представлены алгоритмы подменяемых функций. Схемы алгоритмов процедур commit_kmem_cache_alloc, commit_kmem_cache_free, commit_kmem_cache_destroy представлены на рисунках 2.12 – 2.14.

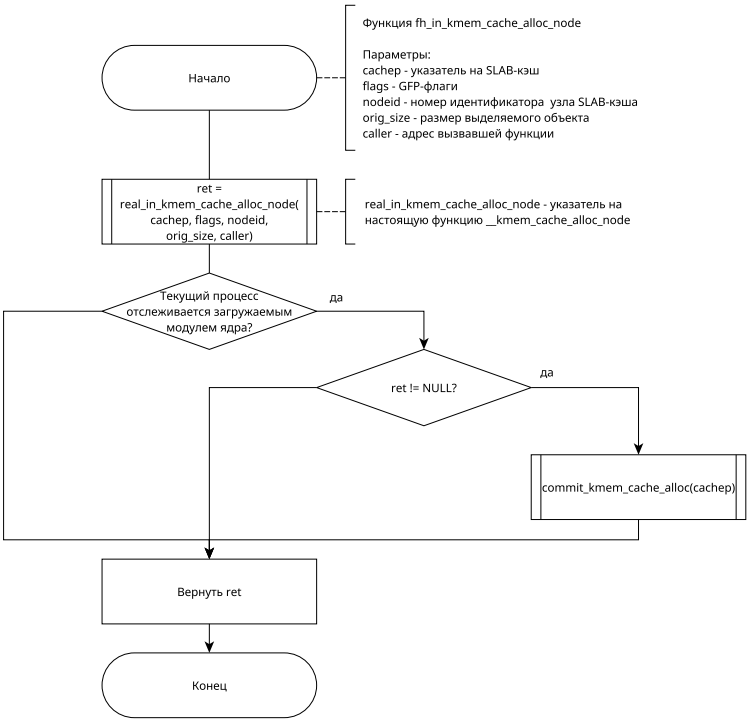

Рисунок 2.7 — Схема алгоритма подменяемой функции для вызова __kmem_cache_alloc_node

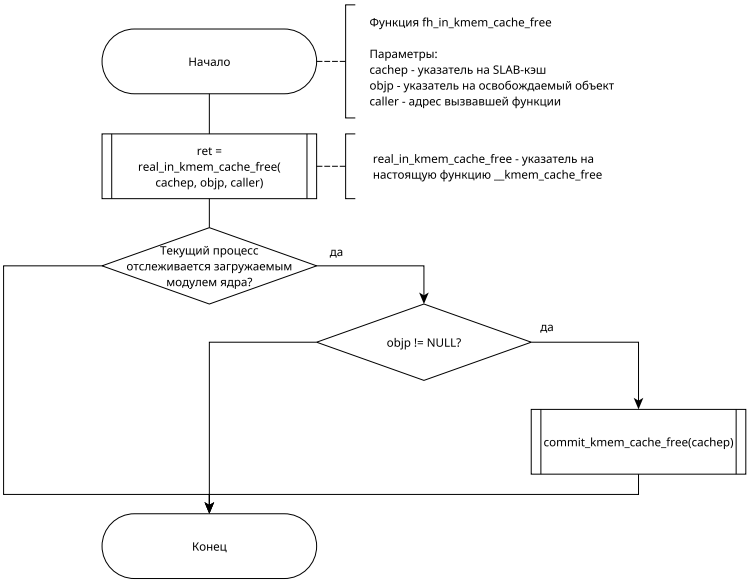

Рисунок 2.8 — Схема алгоритма подменяемой функции для вызова __kmem_cache_free

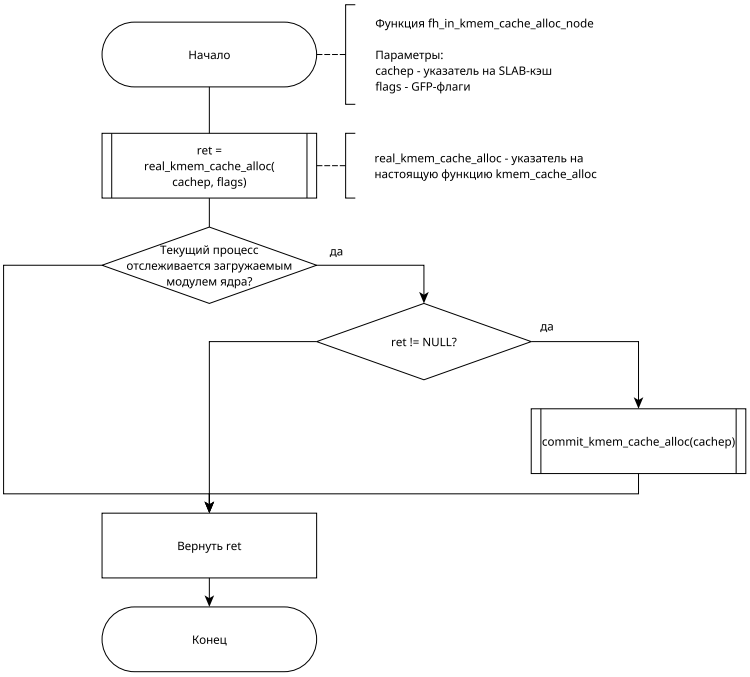

Рисунок 2.9 — Схема алгоритма подменяемой функции для вызова kmem_cache_alloc

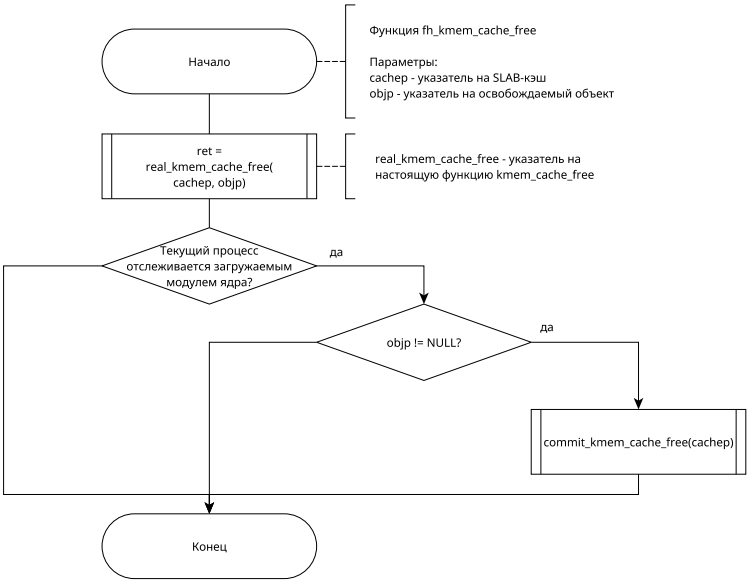

Рисунок 2.10 — Схема алгоритма подменяемой функции для вызова kmem_cache_free

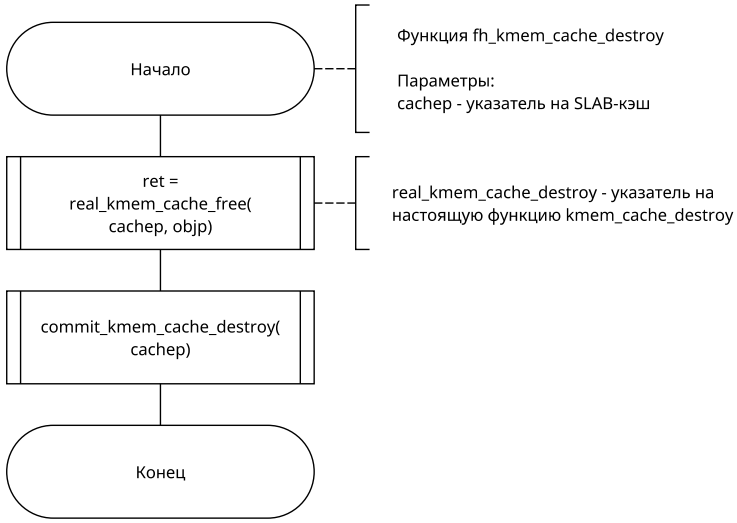

Рисунок 2.11 — Схема алгоритма подменяемой функции для вызова kmem_cache_destroy

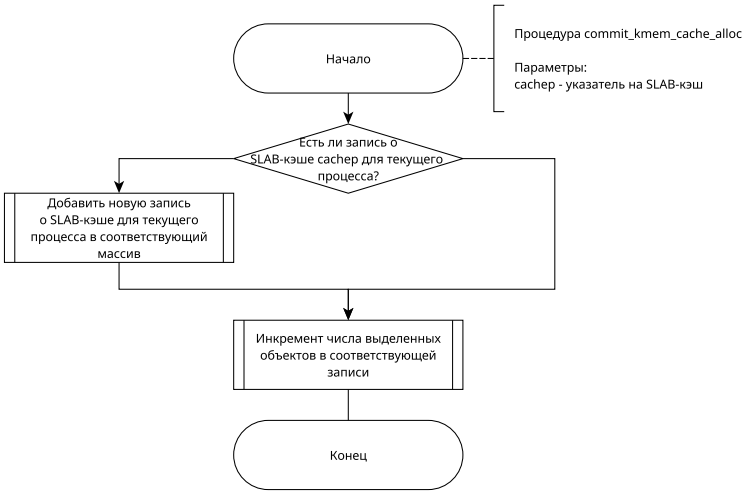

Рисунок 2.12 — Схема алгоритма процедуры commit_kmem_cache_alloc

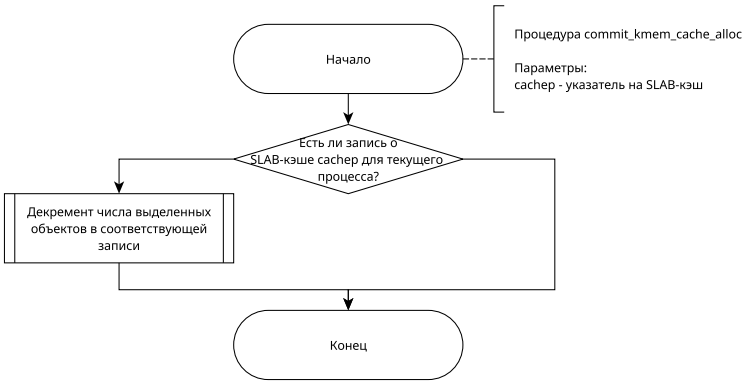

Рисунок 2.13 — Схема алгоритма процедуры commit_kmem_cache_free

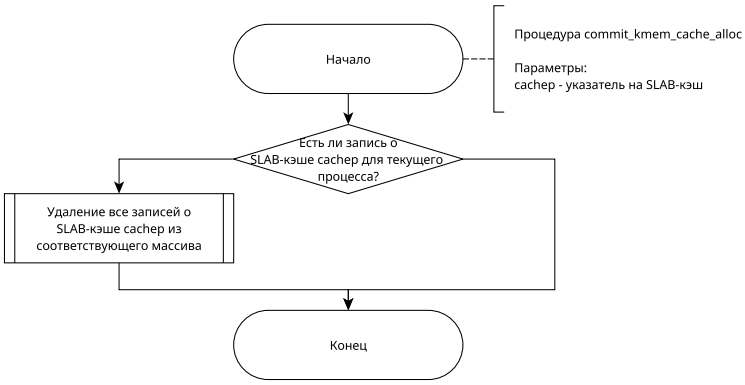

Рисунок 2.14 — Схема алгоритма процедуры commit_kmem_cache_destroy

Вывод

Были разработаны последовательности преобразований в загружаемом модуле ядре для мониторинга использования SLAB-кэша процессами, алгоритмы загрузки и выгрузки разрабатываемого загружаемого модуля ядра, алгоритмы перехвата функции, чтения и записи для файла в виртуальной файловой системе proc и алгоритмы необходимых подменяемых функций.

3 Технологическая часть

3.1 Выбор языка и среды программирования

В качестве языка программирования был выбран язык C, поскольку для данного языка в операционной системе Linux имеется библиотека для разработки загружаемых модулей ядра.

В качестве среды разработки был выбран графический редактор Visual Studio Code.

3.2 Описание структур

Для хранения информации об используемых процессами SLAB-кэшах используются структуры struct proc_slab_info и struct proc_slab_info_array, объявления которых представлены в листингах 3.1 и 3.2 соответственно.

Листинг 3.1 — Объявление структуры struct proc_slab_info

```
struct proc_slab_info
{
  pid_t pid;                /* PID процесса */
  const char *cache_name;   /* имя кэша SLAB */
  size_t num_objs;          /* число объектов в кэше SLAB,
                               выделенных процессом */
  size_t obj_size;          /* размер объекта кэша SLAB */
  size_t objs_per_slab;     /* число объектов в одном slab */
  size_t pages_per_slab;    /* число страниц в одном slab */
};
```

Листинг 3.2 — Объявление структуры struct proc_slab_info_array

```
struct proc_slab_info_array
{
  size_t buf_size;             /* размер буфера, выделенного под массив */
  size_t length;               /* число элементов в массиве */
  struct proc_slab_info *arr; /* массив с информацией об используемых
                                 процессами кэшах SLAB */
};
```

Для хранения PID отслеживаемых процессов используется struct traced_pids_array, объявление которой представлено в листинге 3.3.

Листинг 3.3 — Объявление структуры struct traced_pids_array

```
struct traced_pids_array
{
  size_t buf_size;  /* размер буфера, выделенного под массив */
  size_t length;    /* число элементов в массиве */
  pid_t *arr;       /* массив PID процессов */
};
```

Для осуществления перехвата необходимых функций используется структура struct ftrace_hook, объявление которой представлено в листинге 3.4.

Листинг 3.4 — Объявление структуры struct ftrace_hook

```
struct ftrace_hook
{
  const char *name;        /* имя перехватываемой функции */
  void *function;          /* указатель на функцию, вызываемую
                              вместо перехваченной функции */
  void *original;          /* указатель на оригинальную функцию */
  unsigned long address;   /* адрес оригинальной функции */
  struct ftrace_ops ops;   /* используется для получения объекта
                              структуры struct ftrace_hook
                              в функции обратного вызова с помощью
                              макроса container_of */
};
```

3.3 Реализация загружаемого модуля ядра

Исходный код загружаемого модуля ядра представлен в приложении А.

В листингах 3.5 и 3.6 представлены функции загрузки и выгрузки загружаемого модуля ядра соответственно.

Листинг 3.5 — Функция загрузки ядра

```
static int __init psm_md_init(void)
{
  printk(KERN_INFO "psm_md: loading module\n");
  if (init_tr_pid_array() != 0)
  {
    printk(
      KERN_CRIT "psm_md: cannot allocate memory for traced pids array (
   size %zu)\n",
      DEFAULT_TRACED_PIDS_COUNT * sizeof(pid_t)
    );
    printk(KERN_CRIT "psm_md: module loading has been failed\n");
    return -ENOMEM;
```

```
  }
  if (init_psi_array() != 0)
  {
    printk(
      KERN_CRIT "psm_md: cannot allocate memory for proc slab info
   array (size %zu)\n",
      DEFAULT_TRACED_PIDS_COUNT * sizeof(struct proc_slab_info)
    );
    printk(KERN_CRIT "psm_md: module loading has been failed\n");
    free_traced_pid_array();
    return -ENOMEM;
  }
  static const struct proc_ops fops =
  {
    .proc_open = psm_open,
    .proc_read = psm_read,
    .proc_write = psm_write,
    .proc_release = psm_release
  };
  if (!proc_create(PROC_FILE_NAME, 0x666, NULL, &fops))
  {
    printk(KERN_CRIT "psm_md: cannot create %s file in proc\n",
   PROC_FILE_NAME);
    free_traced_pid_array();
    free_proc_slab_info_array();
    return -EFAULT;
  }
  int rc = fh_install_hooks(hooks, 5);
  if (rc != 0)
  {
    printk(KERN_CRIT "psm_md: cannot hook\n");
    printk(KERN_CRIT "psm_md: module loading has been failed\n");
    free_traced_pid_array();
    free_proc_slab_info_array();
    return -ENOMEM;
  }
  printk(KERN_INFO "psm_md: module has been loaded\n");
  return 0;
}
```

Листинг 3.6 — Функция выгрузки ядра

```
static void __exit psm_md_exit(void)
{
  printk(KERN_INFO "psm_md: unloading module\n");
  printk(KERN_INFO "psm_md: removing hooks\n");
  fh_remove_hooks(hooks, 5);
  printk(KERN_INFO "psm_md: removing proc file\n");
  remove_proc_entry(PROC_FILE_NAME, NULL);
  printk(KERN_INFO "psm_md: freeing pids and psi arrays\n");
  free_traced_pid_array();
  free_proc_slab_info_array();
  printk(KERN_INFO "psm_md: module has been unloaded\n");
}
```

В листингах 3.7 – 3.11 представлены оберточные функции для __kmem_cache_alloc_node, __kmem_cache_free, kmem_cache_alloc, kmem_cache_free и kmem_cache_destroy соответственно.

Листинг 3.7 — Оберточная функция для __kmem_cache_alloc_node

```
static void *( *real_in_kmem_cache_alloc_node)(
  struct kmem_cache *,
  gfp_t,
  int,
  size_t,
  unsigned long
);

static void *fh_in_kmem_cache_alloc_node(
  struct kmem_cache *cachep,
  gfp_t flags,
  int nodeid,
  size_t orig_size,
  unsigned long caller
)
{
  void *ret = real_in_kmem_cache_alloc_node(cachep, flags, nodeid,
   orig_size, caller);
  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched __kmem_cache_alloc_node call from
    %d\n", current->pid);
    if (ret)
      commit_kmem_cache_alloc(cachep);
  }
  return ret;
}
```

Листинг 3.8 — Оберточная функция для __kmem_cache_free

```
static void ( *real_in_kmem_cache_free)(
  struct kmem_cache *,
  void *,
  unsigned long
);

static void fh_in_kmem_cache_free(
  struct kmem_cache *cachep,
  void *objp,
  unsigned long caller
)
{
  real_in_kmem_cache_free(cachep, objp, caller);
  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched __kmem_cache_free call from %d\n"
   , current->pid);
    if (objp)
      commit_kmem_cache_free(cachep);
  }
}
```

Листинг 3.9 — Оберточная функция для kmem_cache_alloc

```
static void *( *real_kmem_cache_alloc)(struct kmem_cache *, gfp_t);

static void *fh_kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags
   )
{
  void *ret = real_kmem_cache_alloc(cachep, flags);
  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched kmem_cache_alloc call from %d\n",
    current->pid);

    if (ret)
      commit_kmem_cache_alloc(cachep);
  }
  return ret;
}
```

Листинг 3.10 — Оберточная функция для kmem_cache_free

```
static void ( *real_kmem_cache_free)(struct kmem_cache *, void *);

static void fh_kmem_cache_free(struct kmem_cache *cachep, void *objp)
{
  real_kmem_cache_free(cachep, objp);
  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched kmem_cache_free call from %d\n",
   current->pid);
    if (objp)
      commit_kmem_cache_free(cachep);
  }
}
```

Листинг 3.11 — Оберточная функция для kmem_cache_destroy

```
static void ( *real_kmem_cache_destroy)(struct kmem_cache *);

static void fh_kmem_cache_destroy(struct kmem_cache *cachep)
{
  real_kmem_cache_destroy(cachep);
  printk(KERN_INFO "psm_md: catched kmem_cache_destroy call from %d\n",
    current->pid);
  if (cachep)
    commit_kmem_cache_destroy(cachep);
}
```

Процедуры commit_kmem_cache_alloc, commit_kmem_cache_free и commit_kmem_cache_destroy представлены в листингах 3.12,3.13 и 3.14 соответственно.

Листинг 3.12 — Процедура commit_kmem_cache_alloc

```
static void commit_kmem_cache_alloc(const struct kmem_cache *const
   cachep)
{
  const char *cache_name = get_kmem_cache_name(cachep);
  int ind = proc_slab_info_ind(current->pid, cache_name);
  if (ind == -1)
  {
    struct proc_slab_info new_psi =
    {
      .pid = current->pid,
      .cache_name = cache_name,
      .num_objs = 0,
      .obj_size = get_kmem_cache_object_size(cachep),
```

```
      .objs_per_slab = get_kmem_cache_objs_per_slab(cachep),
      .pages_per_slab = get_kmem_cache_pages_per_slab(cachep)
    };
    if (add_proc_slab_info(&new_psi) != 0)
      return;
    ind = psi_array.length - 1;
  }
  ++psi_array.arr[ind].num_objs;
}
```

Листинг 3.13 — Процедура commit_kmem_cache_free

```
static void commit_kmem_cache_free(const struct kmem_cache *const
   cachep)
{
  const char *cache_name = get_kmem_cache_name(cachep);
  int ind = proc_slab_info_ind(current->pid, cache_name);
  if (ind != -1 && psi_array.arr[ind].num_objs > 0)
    --psi_array.arr[ind].num_objs;
}
```

Листинг 3.14 — Процедура commit_kmem_cache_destroy

```
static void commit_kmem_cache_destroy(struct kmem_cache *cachep)
{
  size_t i = 0;
  while (i < psi_array.length)
  {
    if (strcmp(psi_array.arr[i].cache_name, get_kmem_cache_name(cachep)
   ) == 0)
    {
      for (size_t j = i; j < psi_array.length; ++j)
        psi_array.arr[j] = psi_array.arr[i + 1];
      --psi_array.length;
    }
    else
      ++i;
  }
}
```

В листингах 3.15 и 3.16 представлены функции для чтения и записи в файл в виртуальной файловой системе proc соответственно.

Листинг 3.15 — Функция psm_read

```
static ssize_t psm_read(struct file *file, char __user *buf, size_t len
   , loff_t *fPos)
{
  printk(KERN_INFO "psm_md: read has been called (PID %d)\n", current->
   pid);
  if ( *fPos >= psi_array.length) return 0;
  char line_buf[500] = {0};
  pid_t pid = psi_array.arr[*fPos].pid;
  const char *cache_name = psi_array.arr[*fPos].cache_name;
  size_t num_objs = psi_array.arr[*fPos].num_objs;
  size_t objs_per_slab = psi_array.arr[*fPos].objs_per_slab;
  size_t pages_per_slab = psi_array.arr[*fPos].pages_per_slab;
  size_t obj_size = psi_array.arr[*fPos].obj_size;
  size_t slabs_count = num_objs / objs_per_slab + (num_objs %
   objs_per_slab > 0);
  size_t total_pages = slabs_count * pages_per_slab;
  if ( *fPos == 0)
    sprintf(
      line_buf,
      "pid   "
      "name                "
      "num_objs   "
      "obj_size   "
      "objs_per_slab "
      "pages_per_slab "
      "total_pages\n"
      "%-5d %-20s %-10zu %-10zu %-13zu %-15zu %-11zu\n",
      pid, cache_name,
      num_objs, obj_size,
      objs_per_slab, pages_per_slab,
      total_pages
    );
  else
    sprintf(
      line_buf,
      "%-5d %-20s %-10zu %-10zu %-13zu %-15zu %-11zu\n",
      pid, cache_name,
      num_objs, obj_size,
      objs_per_slab, pages_per_slab,
      total_pages
    );
  if (copy_to_user(buf, line_buf, strlen(line_buf) + 1) == -1)
  {
    printk(KERN_ERR "psm_md: copy_to_user error\n");
    return -EFAULT;
  }
  ++( *fPos);
  return strlen(line_buf) + 1;
}
```

Листинг 3.16 — Функция psm_write

```
static ssize_t psm_write(struct file *file, const char __user *buf,
   size_t len, loff_t *fPos)
{
  printk(KERN_INFO "psm_md: write has been called (PID %d)\n", current
   ->pid);
  char tmp_buf[10] = {0};
  if (copy_from_user(tmp_buf, buf, len) == -1)
  {
    printk(KERN_ERR "psm_md: copy_from_user error (PID %d)\n", current
   ->pid);
    return -EFAULT;
  }
  pid_t pid = -1;
  printk(KERN_INFO "psm_md: parsing input string (PID %d)\n", current->
   pid);
  if (tmp_buf[0] == 'r')
  {
    int rc = kstrtol(tmp_buf + 1, 10, (long *)&pid);
    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: invalid pid\n");
      return -EIO;
    }
    printk(KERN_INFO "psm_md: removing pid %d (PID %d)\n", pid, current
   ->pid);
    remove_traced_pid(pid);
  }
  else
  {
    int rc = kstrtol(tmp_buf, 10, (long *)&pid);
    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: invalid pid\n");
      return -EIO;
    }
    printk(KERN_INFO "psm_md: adding pid %d (PID %d)\n", pid, current->
   pid);
    rc = add_traced_pid(pid);
    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: error while adding traced pid\n");
      return -ENOMEM;
    }
  }
  return len;
}
```

В листинге 3.17 представлено содержимое make-файла для сборки загружаемого модуля ядра.

Листинг 3.17 — Make-файл для сборки загружаемого модуля ядра

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

obj-m := psm_md.o

default:
  make -C $(KDIR) M=$(PWD) modules
clean:
  @rm -f *.o .*.cmd .*.flags *.mod.c *.order *.mod *.ko *.symvers
  @rm -f .*.*.cmd *~ *.*~ TODO.* .*.d
  @rm -fR .tmp*
  @rm -rf .tmp_versions
  disclean: clean
  @rm *.ko *.symvers
```

Вывод

Были разработаны реализации алгоритмов загрузки и выгрузки разрабатываемого загружаемого модуля ядра, алгоритмы перехвата функции, чтения и записи для файла в виртуальной файловой системе proc и алгоритмы необходимых подменяемых функций. Был разработан make-файл для сборки загружаемого модуля ядра.

4 Исследовательская часть

Для работы с разработанным загружаемым модулем ядра была использована электронная вычислительная машина, обладающая следующими характеристиками:

— операционная система Manjaro Linux x86_64 [9];

— процессор Intel i7-10510U 4.900 ГГц [10];

— оперативная память DDR4, 2400 МГц, 8 ГБ [11].

Для проверки работы разработанного программного обеспечения был реализован загружаемый модуль ядра exp_md, осуществляющий создание и уничтожение SLAB-кэшей, а также выделение и освобождение объектов в SLAB-кэшах. Исходный код загружаемого модуля ядра exp_md представлен в приложении Б.

Во время загрузки загружаемого модуля ядра exp_md в SLAB-кэше выделяются объекты следующих структур:

— struct s1,

— struct s2,

— struct s3.

Объявления данных структур представлены в листинге 4.1.

Листинг 4.1 — Структуры для проведения проверки работы загружаемого модуля ядра для мониторинга SLAB-кэша

```
struct s1
{
  int a;
  char b;

  struct file arr[100];
};

struct s2
{
  int a;
  char b;
};

struct s3
{
  size_t a;
  size_t b;
  size_t c;
  size_t d;

  struct file f;
};
```

Для объектов выше описанных структур создаются SLAB-кэши s1_cache, s2_cache и

s3_cache соответственно.

На рисунках 4.1 – 4.6 представлены данные об использовании SLAB-кэша во время инициализации загружаемого модуля ядра exp_md.

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	12	16	256	1	1
3669	kmem_cache	3	224	32	2	2
3669	kernfs_node_cache	93	128	32	1	3
3669	kmalloc-1k	1	1024	32	8	8
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	1	128	32	1	1
3669	kmalloc-64	3	64	64	1	1
3669	s1_cache	200	23208	1	6	1200
3669	s2_cache	200	8	256	1	1
3669	s3_cache	200	264	30	2	14

Рисунок 4.1 — Данные об использовании SLAB-кэша после выделения 200 объектов структуры struct s1, 200 объектов структуры struct s2 и 200 объектов структуры struct s3

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	12	16	256	1	1
3669	kmem_cache	3	224	32	2	2
3669	kernfs_node_cache	93	128	32	1	3
3669	kmalloc-1k	1	1024	32	8	8
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	1	128	32	1	1
3669	kmalloc-64	3	64	64	1	1
3669	s1_cache	1000	23208	1	6	6000
3669	s2_cache	1000	8	256	1	4
3669	s3_cache	1000	264	30	2	68

Рисунок 4.2 — Данные об использовании SLAB-кэша после выделения 800 объектов структуры struct s1, 800 объектов структуры struct s2 и 800 объектов структуры struct s3

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	12	16	256	1	1
3669	kmem_cache	3	224	32	2	2
3669	kernfs_node_cache	93	128	32	1	3
3669	kmalloc-1k	1	1024	32	8	8
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	1	128	32	1	1
3669	kmalloc-64	4	64	64	1	1
3669	s1_cache	1000	23208	1	6	6000
3669	s2_cache	1000	8	256	1	4
3669	s3_cache	1000	264	30	2	68
3669	kmalloc-32	1	32	128	1	1
3669	kmalloc-8	1500	8	512	1	3

Рисунок 4.3 — Данные об использовании SLAB-кэша после выделения 1500 объектов структуры struct s2 с помощью системного вызова kmalloc

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	12	16	256	1	1
3669	kmem_cache	3	224	32	2	2
3669	kernfs_node_cache	93	128	32	1	3
3669	kmalloc-1k	1	1024	32	8	8
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	1	128	32	1	1
3669	kmalloc-64	4	64	64	1	1
3669	s1_cache	1000	23208	1	6	6000
3669	s2_cache	1000	8	256	1	4
3669	s3_cache	1000	264	30	2	68
3669	kmalloc-32	1	32	128	1	1
3669	kmalloc-8	0	8	512	1	0

Рисунок 4.4 — Данные об использовании SLAB-кэша после освобождения 1500 объектов структуры struct s2 с помощью системного вызова kfree

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	12	16	256	1	1
3669	kmem_cache	3	224	32	2	2
3669	kernfs_node_cache	93	128	32	1	3
3669	kmalloc-1k	1	1024	32	8	8
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	1	128	32	1	1
3669	kmalloc-64	4	64	64	1	1
3669	s1_cache	0	23208	1	6	0
3669	s2_cache	0	8	256	1	0
3669	s3_cache	0	264	30	2	0
3669	kmalloc-32	1	32	128	1	1
3669	kmalloc-8	0	8	512	1	0

Рисунок 4.5 — Данные об использовании SLAB-кэша после освобождения 1000 объектов структуры struct s1, 1000 объектов структуры struct s2 и 1000 объектов структуры struct s3

pid	name	num_objs	obj_size	objs_per_slab	pages_per_slab	total_pages
3669	kmalloc-16	1	16	256	1	1
3669	kmem_cache	0	224	32	2	0
3669	kernfs_node_cache	0	128	32	1	0
3669	kmalloc-1k	0	1024	32	8	0
3669	radix_tree_node	1	576	28	4	4
3669	kmalloc-128	0	128	32	1	0
3669	kmalloc-64	1	64	64	1	1
3669	kmalloc-4k	0	4096	8	8	0
3669	skbuff_head_cache	0	232	32	2	0
3669	kmalloc-96	1	96	42	1	1
3669	kmalloc-32	0	32	128	1	0
3669	sigqueue	1	80	51	1	1

Рисунок 4.6 — Данные об использовании SLAB-кэша после уничтожения кэшей s1_cache, s2_cache и s3_cache

4.1 Вывод

Была проведена проверка работы разработанного загружаемого модуля ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux. Реализованный

загружаемый модуль ядра работает исправно.

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы был разработан загружаемый модуль ядра для мониторинга использования SLAB процессамиа в операционной системе Linux.

Была поставлена задача по разработке загружаемого модуля ядра для отслеживания использования SLAB-кэша процессами в операционной системе Linux. Был проведен анализ распределителей SLAB и SLUB и API для работы с ними, механизмы перехвата функций kprobes и ftrace, виртуальной файловой системы proc. Для перехвата функции был выбран фреймворк ftrace, поскольку он позволяет перехватит любую функцию по ее имени, загружается в ядро динамически и имеет задокументированный API.

Были разработаны последовательности преобразований в загружаемом модуле ядре для мониторинга использования SLAB-кэша процессами, алгоритмы загрузки и выгрузки разрабатываемого загружаемого модуля ядра, алгоритмы перехвата функции, чтения и записи для файла в виртуальной файловой системе proc и алгоритмы необходимых подменяемых функций.

Были разработаны реализации алгоритмов загрузки и выгрузки разрабатываемого загружаемого модуля ядра, алгоритмы перехвата функции, чтения и записи для файла в виртуальной файловой системе proc и алгоритмы необходимых подменяемых функций. Был разработан make-файл для сборки загружаемого модуля ядра.

Была проведена проверка работы разработанного загружаемого модуля ядра для мониторинга использования SLAB-кэша процессами в операционной системе Linux. Реализованный загружаемый модуль ядра работает исправно.

Были выполнены следующие задачи:

— анализ и выбор методов и средств реализации загружаемого модуля ядра;

— разработка структур и алгоритмов, необходимых для работы загружаемого модуля ядра;

— анализ результатов работы разработанного загружаемого модуля ядра.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Slab Allocator [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/gorman/html/understand/understand011.html (дата обращения 10.02.24)

2. slob, slab, slub of linux kernel — Programmer Sought [Электронный ресурс] — Режим доступа: https://www.programmersought.com/article/11795979942/ (дата обращения 10.02.24)

3. Memory Management APIs — The Linux Kernel documentation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/html/latest/core-api/mm-api.html (дата обращения 11.02.24)

4. Physical Page Allocation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/gorman/html/understand/understand009.html (дата обращения 11.02.24)

5. Kernel Probes (Kprobes) — The Linux Kernel documentation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/html/latest/trace/kprobes.html (дата обращения 18.02.24)

6. Using ftrace to hook to functions — The Linux Kernel documentation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/html/latest/trace/ftrace-uses.html (дата обращения 18.02.24)

7. ftrace — Function Tracer — The Linux Kernel documentation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/html/latest/trace/ftrace.html (дата обращения 18.02.24)

8. The /proc Filesystem — The Linux Kernel documentation [Электронный ресурс] — Режим доступа: https://www.kernel.org/doc/html/latest/filesystems/proc.html (дата обращения 18.02.24)

9. Manjaro [Электронный ресурс] — Режим доступа: https://manjaro.org/ (дата обращения 01.03.24)

10. Intel Core i7-10510U Processor [Электронный ресурс] — Режим доступа: https://www.intel.com/content/www/us/en/products/sku/196449/intel-core-i710510u-processor-8m-cache-up-to-4-90-ghz/downloads.html#!=www.intel.com-# (дата обращения 01.03.24)

11. HP ProBook 430 G7 Notebook PC Specifications [Электронный ресурс] — Режим доступа: https://support.hp.com/my-en/document/c06469987 (дата обращения 01.03.24)

Приложение А

Исходный код загружаемого модуля ядра для мониторинга использования SLAB-кэша конкретным процессом

```
#include <linux/slab.h>
#include <linux/mutex.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/ftrace.h>
#include <linux/proc_fs.h>
#include <linux/kprobes.h>
#include <linux/moduleparam.h>

#ifdef CONFIG_SLAB
#include <linux/slab_def.h>
#endif

#ifdef CONFIG_SLUB
/*
typedefs and defines from mm/slab.h for SLUB
struct kmem_cache from linux/slub_def.h
*/
typedef u128 freelist_full_t;

typedef union {
  struct {
    void *freelist;
    unsigned long counter;
  };
  freelist_full_t full;
} freelist_aba_t;

/*
defines from mm/slub.h
*/
#define OO_SHIFT 16
#define OO_MASK ((1 << OO_SHIFT) - 1)

#include <linux/slub_def.h>
#endif
```

```
#define PROC_FILE_NAME "procslabinfo"

#define DEFAULT_TRACED_PIDS_COUNT 10
#define DEFAULT_PROC_SLAB_INFO_COUNT 50

MODULE_LICENSE("GPL");

struct traced_pids_array
{
  size_t buf_size;
  size_t length;

  pid_t *arr;
};

struct proc_slab_info
{
  pid_t pid;

  const char *cache_name;

  size_t num_objs;
  size_t obj_size;

  size_t objs_per_slab;
  size_t pages_per_slab;
};

struct proc_slab_info_array
{
  size_t buf_size;
  size_t length;

  struct proc_slab_info *arr;
};

struct ftrace_hook
{
  const char *name;
  void *function;
```

```
  void *original;

  unsigned long address;
  struct ftrace_ops ops;
};

static struct traced_pids_array tr_pid_array = {0, 0, NULL};
static struct proc_slab_info_array psi_array = {0, 0, NULL};

static unsigned long lookup_name(const char *name)
{
  struct kprobe kp =
  {
    .symbol_name = name
  };

  unsigned long retval;

  if (register_kprobe(&kp) < 0)
    return 0;

  retval = (unsigned long) kp.addr;

  unregister_kprobe(&kp);

  return retval;
}

/**********************************
   struct kmem_cache data access
**********************************/

static const char *get_kmem_cache_name(const struct kmem_cache *const
   cachep)
{
#ifdef CONFIG_SLAB
  return cachep->name;
#endif

#ifdef CONFIG_SLUB
  return cachep->name;
```

```
#endif
}

static size_t get_kmem_cache_object_size(const struct kmem_cache *const
    cachep)
{
#ifdef CONFIG_SLAB
  return (size_t)cachep->object_size;
#endif

#ifdef CONFIG_SLUB
  return (size_t)cachep->object_size;
#endif
}

// TODO: objects count sometimes is very big
static size_t get_kmem_cache_objs_per_slab(const struct kmem_cache *
   const cachep)
{
#ifdef CONFIG_SLAB
  return cachep->num;
#endif

#ifdef CONFIG_SLUB
  return (size_t)(cachep->oo.x) & OO_MASK;
#endif
}

static size_t get_kmem_cache_pages_per_slab(const struct kmem_cache *
   const cachep)
{
#ifdef CONFIG_SLAB
  return (size_t)int_pow(2, cachep->gfporder);
#endif

#ifdef CONFIG_SLUB
  size_t objs_per_slab = get_kmem_cache_objs_per_slab(cachep);
  size_t slab_size = objs_per_slab * (size_t)cachep->size;

  return slab_size / PAGE_SIZE + (slab_size % PAGE_SIZE > 0);
#endif
```

```
}

/**********************************
   proc_slab_info_array operations
**********************************/

static int init_psi_array(void)
{
  if (!(psi_array.arr = kmalloc(DEFAULT_PROC_SLAB_INFO_COUNT * sizeof(
   struct proc_slab_info), GFP_KERNEL)))
    return -ENOMEM;

  psi_array.buf_size = DEFAULT_PROC_SLAB_INFO_COUNT * sizeof(struct
   proc_slab_info);

  return 0;
}

static int proc_slab_info_ind(const pid_t pid, const char *cache_name)
{
  int i = 0;
  int found = 0;

  while (i < psi_array.length && !found)
  {
    bool pids_are_equal = psi_array.arr[i].pid == pid;
    bool cache_names_are_equal = strcmp(cache_name, psi_array.arr[i].
   cache_name) == 0;

    if (pids_are_equal && cache_names_are_equal)
      found = 1;
    else
      ++i;
  }

  return i < psi_array.length ? i : -1;
}

static int add_proc_slab_info(const struct proc_slab_info *const
   new_psi)
{
```

```
  if (psi_array.length >= psi_array.buf_size)
  {
    struct proc_slab_info *new_arr = krealloc(psi_array.arr, 2 *
   psi_array.buf_size * sizeof(struct proc_slab_info), GFP_KERNEL);

    if (!new_arr)
    {
      printk(KERN_ERR "psm_md: cannot reallocate memory for proc slab
   info array\n");

      return -ENOMEM;
    }

    psi_array.buf_size *= 2 * sizeof(struct proc_slab_info);
    psi_array.arr = new_arr;
  }

  psi_array.arr[psi_array.length++] = *new_psi;

  return 0;
}

static void remove_proc_slab_info_by_pid(const pid_t pid)
{
  for (int i = 0; i < psi_array.length; ++i)
  {
    if (psi_array.arr[i].pid == pid)
    {
      for (int j = i; j < psi_array.length - 1; ++j)
        psi_array.arr[j] = psi_array.arr[j + 1];

      --psi_array.length;
    }
    else
      ++i;
  }
}

static void free_proc_slab_info_array(void)
{
  kfree(psi_array.arr);
```

```
  psi_array.arr = NULL;
}

/**********************************
    traced_pids_array operations
**********************************/

static int init_tr_pid_array(void)
{
  if (!(tr_pid_array.arr = kmalloc(DEFAULT_TRACED_PIDS_COUNT * sizeof(
   pid_t), GFP_KERNEL)))
    return -ENOMEM;

  tr_pid_array.buf_size = DEFAULT_TRACED_PIDS_COUNT * sizeof(pid_t);

  return 0;
}

static int traced_pid_ind(const pid_t pid)
{
  int i = 0;

  for (; i < tr_pid_array.length && tr_pid_array.arr[i] != pid; ++i);

  return i < tr_pid_array.length ? i : -1;
}

static void remove_traced_pid(const pid_t pid)
{
  int i = traced_pid_ind(pid);

  if (i != -1)
  {
    remove_proc_slab_info_by_pid(pid);

    for (int j = i; j < tr_pid_array.length - 1; ++j)
      tr_pid_array.arr[j] = tr_pid_array.arr[j + 1];

    --tr_pid_array.length;
  }
}
```

```
static int add_traced_pid(const pid_t pid)
{
  int i = traced_pid_ind(pid);

  if (i == -1)
  {
    printk(KERN_INFO "psm_md: pid %d not found, adding", pid);

    if (tr_pid_array.length >= tr_pid_array.buf_size)
    {
      pid_t *new_arr = krealloc(tr_pid_array.arr, 2 * tr_pid_array.
   buf_size * sizeof(pid_t), GFP_KERNEL);

      if (!new_arr)
      {
        printk(KERN_ERR "psm_md: cannot reallocate memory for traced
   pids array\n");

        return -ENOMEM;
      }

      tr_pid_array.buf_size *= 2 * sizeof(pid_t);
      tr_pid_array.arr = new_arr;
    }

    tr_pid_array.arr[tr_pid_array.length++] = pid;
  }

  return 0;
}

static void free_traced_pid_array(void)
{
  kfree(tr_pid_array.arr);
  tr_pid_array.arr = NULL;
}

/***********************************
              Fortunes
***********************************/
```

```
static int psm_open(struct inode *sp_inode, struct file *sp_file)
{
  printk(KERN_INFO "psm_md: open has been called (PID %d)\n", current->
   pid);

  return 0;
}

static int psm_release(struct inode *sp_inode, struct file *sp_file)
{
  printk(KERN_INFO "psm_md: release has been called (PID %d)\n",
   current->pid);

  return 0;
}

static ssize_t psm_write(struct file *file, const char __user *buf,
   size_t len, loff_t *fPos)
{
  printk(KERN_INFO "psm_md: write has been called (PID %d)\n", current
   ->pid);

  char tmp_buf[10] = {0};

  if (copy_from_user(tmp_buf, buf, len) == -1)
  {
    printk(KERN_ERR "psm_md: copy_from_user error (PID %d)\n", current
   ->pid);

    return -EFAULT;
  }

  pid_t pid = -1;

  printk(KERN_INFO "psm_md: parsing input string (PID %d)\n", current->
   pid);

  if (tmp_buf[0] == 'r')
  {
    int rc = kstrtol(tmp_buf + 1, 10, (long *)&pid);
```

```
    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: invalid pid\n");

      return -EIO;
    }

    printk(KERN_INFO "psm_md: removing pid %d (PID %d)\n", pid, current
   ->pid);

    remove_traced_pid(pid);
  }
  else
  {
    int rc = kstrtol(tmp_buf, 10, (long *)&pid);

    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: invalid pid\n");

      return -EIO;
    }

    printk(KERN_INFO "psm_md: adding pid %d (PID %d)\n", pid, current->
   pid);

    rc = add_traced_pid(pid);

    if (rc != 0)
    {
      printk(KERN_ERR "psm_md: error while adding traced pid\n");

      return -ENOMEM;
    }
  }

  return len;
}

static ssize_t psm_read(struct file *file, char __user *buf, size_t len
```

```
    , loff_t *fPos)
{
  printk(KERN_INFO "psm_md: read has been called (PID %d)\n", current->
   pid);

  if ( *fPos >= psi_array.length)
  return 0;

  char line_buf[500] = {0};

  pid_t pid = psi_array.arr[*fPos].pid;
  const char *cache_name = psi_array.arr[*fPos].cache_name;
  size_t num_objs = psi_array.arr[*fPos].num_objs;
  size_t objs_per_slab = psi_array.arr[*fPos].objs_per_slab;
  size_t pages_per_slab = psi_array.arr[*fPos].pages_per_slab;
  size_t obj_size = psi_array.arr[*fPos].obj_size;

  size_t slabs_count = num_objs / objs_per_slab + (num_objs %
   objs_per_slab > 0);
  size_t total_pages = slabs_count * pages_per_slab;

  if ( *fPos == 0)
    sprintf(
      line_buf,
      "pid   "
      "name                "
      "num_objs   "
      "obj_size   "
      "objs_per_slab "
      "pages_per_slab "
      "total_pages\n"
      "%-5d %-20s %-10zu %-10zu %-13zu %-15zu %-11zu\n",
      pid,
      cache_name,
      num_objs,
      obj_size,
      objs_per_slab,
      pages_per_slab,
      total_pages
    );
  else
```

```
    sprintf(
      line_buf,
      "%-5d %-20s %-10zu %-10zu %-13zu %-15zu %-11zu\n",
      pid,
      cache_name,
      num_objs,
      obj_size,
      objs_per_slab,
      pages_per_slab,
      total_pages
    );

  if (copy_to_user(buf, line_buf, strlen(line_buf) + 1) == -1)
  {
    printk(KERN_ERR "psm_md: copy_to_user error\n");

    return -EFAULT;
  }

  ++( *fPos);

  return strlen(line_buf) + 1;
}

/**********************************
               ftrace
**********************************/

static void notrace ftrace_callback(unsigned long ip, unsigned long
   parent_ip, struct ftrace_ops *ops, struct ftrace_regs *fregs)
{
  struct pt_regs *regs = ftrace_get_regs(fregs);
  struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops)
   ;

  if (!within_module(parent_ip, THIS_MODULE))
    regs->ip = (unsigned long)hook->function;
}

static int fh_resolve_hook_address(struct ftrace_hook *hook)
{
```

```
  hook->address = lookup_name(hook->name);

  if (!hook->address)
  {
    printk(KERN_ERR "psm_md: unresolved symbol (%s) in lookup_name\n",
   hook->name);

    return -ENOENT;
  }

  *((unsigned long*) hook->original) = hook->address;

  return 0;
}

static int fh_install_hook(struct ftrace_hook *hook)
{
  int rc = fh_resolve_hook_address(hook);

  if (rc != 0)
  return rc;

  hook->ops.func = ftrace_callback;
  hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_RECURSION |
    FTRACE_OPS_FL_IPMODIFY;

  rc = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);

  if (rc != 0)
  {
    printk(KERN_ERR "psm_md: ftrace_set_filter_ip failed with code %d\n
   ", rc);

    return rc;
  }

  rc = register_ftrace_function(&hook->ops);

  if (rc != 0)
  {
    printk(KERN_ERR "psm_md: register_ftrace_function failed with code
```

```
   %d\n", rc);

    ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);

    return rc;
  }

  return 0;
}

void fh_remove_hook(struct ftrace_hook *hook)
{
  int rc = unregister_ftrace_function(&hook->ops);

  if (rc != 0)
    printk(KERN_ERR "psm_md: unregister_ftrace_function failed with
   code %d\n", rc);

  rc = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);

  if (rc != 0)
    printk(KERN_ERR "psm_md: ftrace_set_filter_ip failed with code %d\n
   ", rc);
}

int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
{
  int rc;
  size_t i;

  for (i = 0; i < count; i++)
  {
    rc = fh_install_hook(&hooks[i]);

    if (rc != 0)
    {
      while (i != 0)
      fh_remove_hook(&hooks[--i]);

      return rc;
    }
```

```
  }

  return 0;
}

void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
{
  size_t i;

  for (i = 0; i < count; i++)
  fh_remove_hook(&hooks[i]);
}

static void commit_kmem_cache_alloc(
  const struct kmem_cache *const cachep
)
{
  const char *cache_name = get_kmem_cache_name(cachep);

  int ind = proc_slab_info_ind(current->pid, cache_name);

  if (ind == -1)
  {
    struct proc_slab_info new_psi =
    {
      .pid = current->pid,
      .cache_name = cache_name,
      .num_objs = 0,
      .obj_size = get_kmem_cache_object_size(cachep),
      .objs_per_slab = get_kmem_cache_objs_per_slab(cachep),
      .pages_per_slab = get_kmem_cache_pages_per_slab(cachep)
    };

    if (add_proc_slab_info(&new_psi) != 0)
      return;

    ind = psi_array.length - 1;
  }

  ++psi_array.arr[ind].num_objs;
}
```

```
static void commit_kmem_cache_free(const struct kmem_cache *const
   cachep)
{
  const char *cache_name = get_kmem_cache_name(cachep);

  int ind = proc_slab_info_ind(current->pid, cache_name);

  if (ind != -1 && psi_array.arr[ind].num_objs > 0)
    --psi_array.arr[ind].num_objs;
}

static void commit_kmem_cache_destroy(struct kmem_cache *cachep)
{
  size_t i = 0;

  while (i < psi_array.length)
  {
    if (strcmp(psi_array.arr[i].cache_name, get_kmem_cache_name(cachep)
   ) == 0)
    {
      for (size_t j = i; j < psi_array.length; ++j)
      psi_array.arr[j] = psi_array.arr[i + 1];

      --psi_array.length;
    }
    else
      ++i;
  }
}

static void *( *real_in_kmem_cache_alloc_node)(
  struct kmem_cache *,
  gfp_t,
  int,
  size_t,
  unsigned long
);

static void *fh_in_kmem_cache_alloc_node(
  struct kmem_cache *cachep,
```

```
  gfp_t flags,
  int nodeid,
  size_t orig_size,
  unsigned long caller
)
{
  void *ret = real_in_kmem_cache_alloc_node(cachep, flags, nodeid,
   orig_size, caller);

  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched __kmem_cache_alloc_node call from
    %d\n", current->pid);

    if (ret)
      commit_kmem_cache_alloc(cachep);
  }

  return ret;
}

static void ( *real_in_kmem_cache_free)(
  struct kmem_cache *,
  void *,
  unsigned long
);

static void fh_in_kmem_cache_free(
  struct kmem_cache *cachep,
  void *objp,
  unsigned long caller
)
{
  real_in_kmem_cache_free(cachep, objp, caller);

  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched __kmem_cache_free call from %d\n"
   , current->pid);

    if (objp)
```

```
      commit_kmem_cache_free(cachep);
  }
}

static void *( *real_kmem_cache_alloc)(struct kmem_cache *, gfp_t);

static void *fh_kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags
   )
{
  void *ret = real_kmem_cache_alloc(cachep, flags);

  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched kmem_cache_alloc call from %d\n",
    current->pid);

    if (ret)
      commit_kmem_cache_alloc(cachep);
  }

  return ret;
}

static void ( *real_kmem_cache_free)(struct kmem_cache *, void *);

static void fh_kmem_cache_free(struct kmem_cache *cachep, void *objp)
{
  real_kmem_cache_free(cachep, objp);

  if (traced_pid_ind(current->pid) != -1)
  {
    printk(KERN_INFO "psm_md: catched kmem_cache_free call from %d\n",
   current->pid);

    if (objp)
      commit_kmem_cache_free(cachep);
  }
}

static void ( *real_kmem_cache_destroy)(struct kmem_cache *);
```

```
static void fh_kmem_cache_destroy(struct kmem_cache *cachep)
{
  real_kmem_cache_destroy(cachep);

  printk(KERN_INFO "psm_md: catched kmem_cache_destroy call from %d\n",
    current->pid);

  if (cachep)
    commit_kmem_cache_destroy(cachep);
}

static struct ftrace_hook hooks[] =
{
  {
    .name = "kmem_cache_alloc",
    .function = fh_kmem_cache_alloc,
    .original = &real_kmem_cache_alloc
  },
  {
    .name = "kmem_cache_free",
    .function = fh_kmem_cache_free,
    .original = &real_kmem_cache_free
  },
  {
    .name = "__kmem_cache_alloc_node",
    .function = fh_in_kmem_cache_alloc_node,
    .original = &real_in_kmem_cache_alloc_node
  },
  {
    .name = "__kmem_cache_free",
    .function = fh_in_kmem_cache_free,
    .original = &real_in_kmem_cache_free
  },
  {
    .name = "kmem_cache_destroy",
    .function = fh_kmem_cache_destroy,
    .original = &real_kmem_cache_destroy
  }
};

/*********************************
```

```
          Module init/exit
**********************************/

static int __init psm_md_init(void)
{
  printk(KERN_INFO "psm_md: loading module\n");

  if (init_tr_pid_array() != 0)
  {
    printk(
    KERN_CRIT "psm_md: cannot allocate memory for traced pids array (
   size %zu)\n",
    DEFAULT_TRACED_PIDS_COUNT * sizeof(pid_t)
    );
    printk(KERN_CRIT "psm_md: module loading has been failed\n");

    return -ENOMEM;
  }

  if (init_psi_array() != 0)
  {
    printk(
    KERN_CRIT "psm_md: cannot allocate memory for proc slab info array
   (size %zu)\n",
    DEFAULT_TRACED_PIDS_COUNT * sizeof(struct proc_slab_info)
    );
    printk(KERN_CRIT "psm_md: module loading has been failed\n");

    free_traced_pid_array();

    return -ENOMEM;
  }

  static const struct proc_ops fops =
  {
    .proc_open = psm_open,
    .proc_read = psm_read,
    .proc_write = psm_write,
    .proc_release = psm_release
  };
```

```
  if (!proc_create(PROC_FILE_NAME, 0x666, NULL, &fops))
  {
    printk(KERN_CRIT "psm_md: cannot create %s file in proc\n",
   PROC_FILE_NAME);

    free_traced_pid_array();
    free_proc_slab_info_array();

    return -EFAULT;
  }

  int rc = fh_install_hooks(hooks, 5);

  if (rc != 0)
  {
    printk(KERN_CRIT "psm_md: cannot hook\n");
    printk(KERN_CRIT "psm_md: module loading has been failed\n");

    free_traced_pid_array();
    free_proc_slab_info_array();

    return -ENOMEM;
  }

  printk(KERN_INFO "psm_md: module has been loaded\n");

  return 0;
}

static void __exit psm_md_exit(void)
{
  printk(KERN_INFO "psm_md: unloading module\n");

  printk(KERN_INFO "psm_md: removing hooks\n");
  fh_remove_hooks(hooks, 5);

  printk(KERN_INFO "psm_md: removing proc file\n");
  remove_proc_entry(PROC_FILE_NAME, NULL);

  printk(KERN_INFO "psm_md: freeing pids and psi arrays\n");
  free_traced_pid_array();
```

```
  free_proc_slab_info_array();

  printk(KERN_INFO "psm_md: module has been unloaded\n");
}

module_init(psm_md_init);
module_exit(psm_md_exit);
```

Приложение Б

Исходный код загружаемого модуля ядра exp_md для проверки работы разработанного программного обеспечения

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/vmalloc.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");

struct s1
{
  int a;
  char b;

  struct file arr[100];
};

struct s2
{
  int a;
  char b;
};

struct s3
{
  size_t a;
  size_t b;
  size_t c;
  size_t d;

  struct file f;
};

void s1_ctor(void *s1)
{}
```

```
void s2_ctor(void *s2)
{}

void s3_ctor(void *s3)
{}

static int __init psm_md_init(void)
{
  printk(KERN_INFO "exp_md: loading module (PID %d)\n", current->pid);

  msleep(20000);

  struct kmem_cache *s1_cache = kmem_cache_create("s1_cache", sizeof(
   struct s1), 0, 0, s1_ctor);
  struct kmem_cache *s2_cache = kmem_cache_create("s2_cache", sizeof(
   struct s2), 0, 0, s2_ctor);
  struct kmem_cache *s3_cache = kmem_cache_create("s3_cache", sizeof(
   struct s3), 0, 0, s3_ctor);

  printk(KERN_INFO "exp_md: created caches\n");

  struct s1 **s1_arr = vmalloc(1000 * sizeof(struct s1 *));
  struct s2 **s2_arr = vmalloc(1000 * sizeof(struct s2 *));
  struct s3 **s3_arr = vmalloc(1000 * sizeof(struct s3 *));

  for (size_t i = 0; i < 200; ++i)
  {
    s1_arr[i] = kmem_cache_alloc(s1_cache, GFP_KERNEL);
    s2_arr[i] = kmem_cache_alloc(s2_cache, GFP_KERNEL);
    s3_arr[i] = kmem_cache_alloc(s3_cache, GFP_KERNEL);
  }

  printk(KERN_INFO "exp_md: (1) allocated 200 s1, 200 s2, 200 s3\n");

  msleep(10000);

  for (size_t i = 200; i < 1000; ++i)
  {
    s1_arr[i] = kmem_cache_alloc(s1_cache, GFP_KERNEL);
    s2_arr[i] = kmem_cache_alloc(s2_cache, GFP_KERNEL);
    s3_arr[i] = kmem_cache_alloc(s3_cache, GFP_KERNEL);
```

```
}

printk(KERN_INFO "exp_md: (2) allocated 800 s1, 800 s2, 800 s3\n");

msleep(10000);

struct s2 **s2_km_arr = vmalloc(1500 * sizeof(struct s2 *));

for (size_t i = 0; i < 1500; ++i)
  s2_km_arr[i] = kmalloc(sizeof(struct s2), GFP_KERNEL);

printk(KERN_INFO "exp_md: 1500 s1 with kmalloc\n");

msleep(10000);

for (size_t i = 0; i < 1500; ++i)
  kfree(s2_km_arr[i]);

printk(KERN_INFO "exp_md: free 1500 s1 with kfree\n");

msleep(10000);

for (size_t i = 0; i < 1000; ++i)
{
  kmem_cache_free(s1_cache, s1_arr[i]);
  kmem_cache_free(s2_cache, s2_arr[i]);
  kmem_cache_free(s3_cache, s3_arr[i]);
}

printk(KERN_INFO "exp_md: free 1000 s1, 1000 s2, 1000 s3\n");

msleep(10000);

kmem_cache_destroy(s1_cache);
kmem_cache_destroy(s2_cache);
kmem_cache_destroy(s3_cache);

vfree(s1_arr);
vfree(s2_arr);
vfree(s3_arr);
vfree(s2_km_arr);
```

```
  printk(KERN_INFO "exp_md: module has been loaded\n");

  return 0;
}

static void __exit psm_md_exit(void)
{
  printk(KERN_INFO "exp_md: unloading module\n");

  printk(KERN_INFO "exp_md: module has been unloaded\n");
}

module_init(psm_md_init);
module_exit(psm_md_exit);
```