



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 ИУ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ИУ7 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:
Мониторинг выделения памяти в SLAB-кеше

Студент ИУ7-74Б
(Группа)

(Подпись, дата) К.А. Рунов
(И.О.Фамилия)

Руководитель курсовой работы

(Подпись, дата) Н.Ю. Рязанова
(И.О.Фамилия)

2025 г.

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)

УТВЕРЖДАЮ
Заведующий кафедрой _____
(Индекс)

(И.О.Фамилия)
« ____ » _____ 20 ____ г.

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине _____ Операционные системы

Студент группы _____ ИУ7-74Б

_____ Рунов Константин Алексеевич
(Фамилия, имя, отчество)

Тема курсовой работы Мониторинг системных вызовов

Направленность КР (учебная, исследовательская, практическая, производственная, др.)
_____ учебная

Источник тематики (кафедра, предприятие, НИР) _____ Кафедра _____

График выполнения работы: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание

Разработать загружаемый модуль ядра для операционной системы GNU/Linux, позволяющий осуществлять мониторинг выделения памяти в адресном пространстве ядра, предоставляющий информацию о запросах выделения физической памяти и выделения памяти в SLAB-кеше.

Оформление курсовой работы:

Расчетно-пояснительная записка на 12-32 листах формата А4, презентация к курсовой работе на 8–16 слайдах

Дата выдачи задания « ____ » _____ 2024 г.

Руководитель курсовой работы

Студент

_____	_____ <u>Н.Ю. Рязанова</u>
(Подпись, дата)	(И.О.Фамилия)
_____	_____ <u>К.А. Рунов</u>
(Подпись, дата)	(И.О.Фамилия)

РЕФЕРАТ

Отчет 59 с., 9 рис., 1 табл., NNIGGERNIGGERNIGGERNIGGERNIGGERNIC
источн., NNIGGERNIGGERNIGGERNIGGERNIGGERNIGGERNIGGERNIGGER
прил.

СОДЕРЖАНИЕ

РЕФЕРАТ	3
ВВЕДЕНИЕ	6
1 Аналитический раздел	7
1.1 Постановка задачи	7
1.2 Выделение памяти в ядре Linux	7
1.2.1 NUMA	7
1.2.2 NUMA, узлы памяти и зоны в Linux	8
1.2.3 Зоны в Linux	10
1.2.4 Структура struct page и массив mem_map	11
1.2.5 SLAB-кэш	14
1.2.6 Структуры struct kmem_cache и struct slab	15
1.3 Функции и системные вызовы	17
1.3.1 alloc_pages	17
1.3.2 kmem_cache_*	19
1.3.3 kmalloc	21
1.3.4 kfree	23
1.4 Анализ способов мониторинга памяти	23
1.4.1 Перехват системных вызовов и хуки ядра	23
1.4.2 Сравнительный анализ подходов	26
1.4.3 Выбор метода мониторинга	26
2 Конструкторский раздел	27
3 Технологический раздел	28
3.1 Средства реализации	28
3.2 Реализация загружаемого модуля ядра	28
3.3 Реализация Makefile	32
3.4 Реализация программы для генерации потоков	33
4 Исследовательский раздел	34
4.1 Технические характеристики	34

4.2	Демонстрация работы ПО	35
4.3	Анализ результатов работы ПО	39
ЗАКЛЮЧЕНИЕ		40
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ		42
ПРИЛОЖЕНИЕ А		43

ВВЕДЕНИЕ

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать загружаемый модуль ядра, позволяющий осуществлять мониторинг выделения памяти в адресном пространстве ядра, предоставляющий информацию о запросах выделения физической памяти и выделения памяти в SLAB-кэше. Для решения поставленной задачи необходимо решить следующие задачи:

- 1) провести обзор способов выделения памяти в ядре Linux;
- 2) провести обзор способов мониторинга выделения памяти;
- 3) провести сравнительный анализ подходов к решению поставленной задачи;
- 4) разработать алгоритмы и привести структуры данных для решения поставленной задачи;
- 5) реализовать загружаемый модуль ядра, решающий поставленную задачу;
- 6) протестировать разработанное программное обеспечение.

1.2 Выделение памяти в ядре Linux

1.2.1 NUMA

Одним из наиболее распространенных понятий в управлении памятью является NUMA (англ. Non-Uniform Memory Access, неравномерный доступ к памяти или Non-Uniform Memory Architecture, архитектура с неравномерной памятью) — архитектура организации компьютерной памяти, используемая в мультипроцессорных системах. Процессор имеет быстрый доступ к локальной памяти через свой контроллер, а также более медленный канал до памяти, подключенной к контроллерам (слотам) других процессоров, реализуемый через шину обмена данными. [1]

На рисунке ниже приведена топология NUMA.

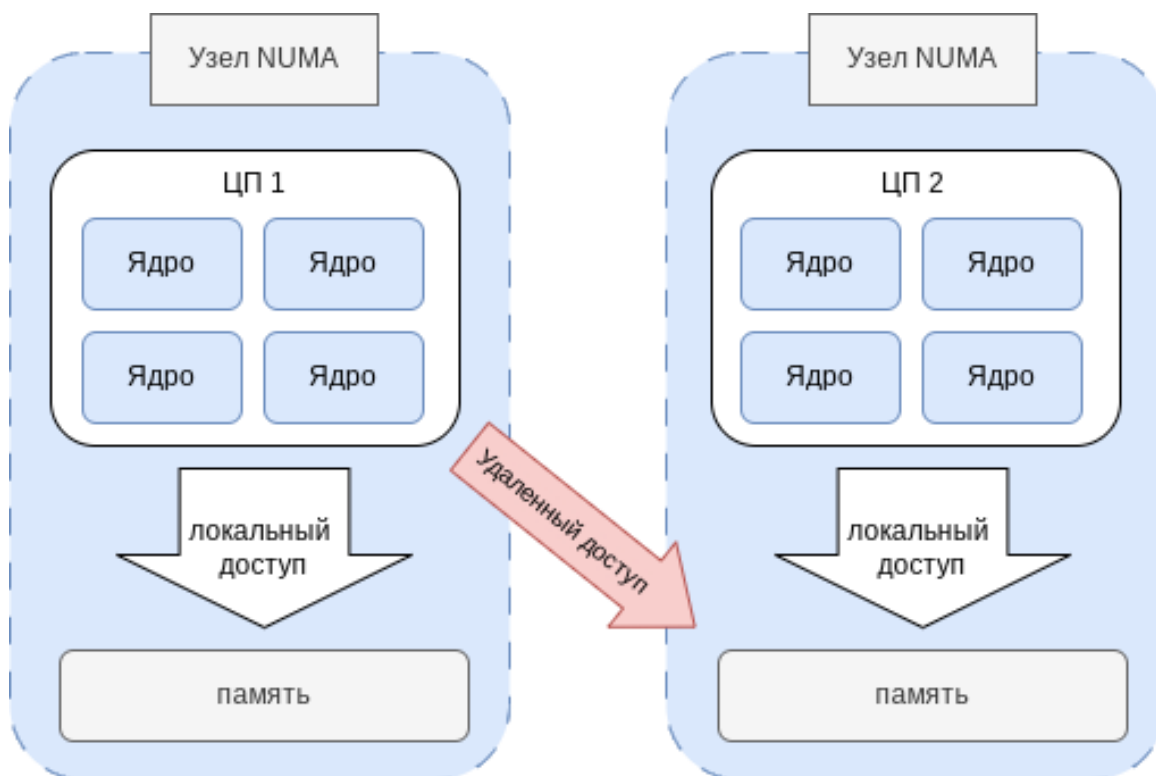


Рисунок 1 – Топология NUMA

То есть, каждый узел NUMA содержит:

- 1) CPU (или несколько ядер);
- 2) Локальную память (RAM);
- 3) Контроллер памяти.

Если процесс выполняется на CPU и запрашивает память из другого NUMA-узла, доступ к ней будет медленнее, чем если бы она была выделена из локального узла. Поэтому ядро Linux старается выделять память из локального узла.

1.2.2 NUMA, узлы памяти и зоны в Linux

В многопроцессорных (англ. multi-core) и многосокетных (англ. multi-socket) системах память может быть организована в банки, доступ к которым имеет разную стоимость в зависимости от их «удалённости» от процессора. Например, может существовать банк памяти, закреплённый за каждым процессором, или банк памяти, предназначенный для DMA, расположенный рядом с периферийными устройствами.

Каждый такой банк памяти называется узлом (англ. node), и в Linux он представлен структурой `struct pglist_data`, даже если архитектура является UMA (англ. Uniform Memory Access, равномерный доступ к памяти). Эта структура всегда используется через `typedef pg_data_t`. Чтобы получить структуру `pg_data_t` для конкретного узла, используется макрос `NODE_DATA(nid)`, где `nid` — это идентификатор (ID) узла.

Ниже приведен листинг структуры `pg_data_t` из ядра Linux версии 6.6.

```
1 typedef struct pglist_data {
2     struct zone node_zones[MAX_NR_ZONES];
3     struct zonelist node_zonelists[MAX_ZONELISTS];
4
5     /* number of populated zones in this node */
6     int nr_zones;
7     ...
8     unsigned long node_start_pfn;
9
10    /* total number of physical pages */
11    unsigned long node_present_pages;
12
13    /* total size of physical page range, including holes */
14    unsigned long node_spanned_pages;
15    int node_id;
16    ...
17    /*
18     * This is a per-node reserve of pages that are not available
19     * to userspace allocations.
20     */
21    unsigned long      totalreserve_pages;
22
23    #ifdef CONFIG_NUMA
24        /*
25         * node reclaim becomes active if more unmapped pages exist.
26         */
27        unsigned long      min_unmapped_pages;
28        unsigned long      min_slab_pages;
29    #endif /* CONFIG_NUMA */
30    ...
31 } pg_data_t;
```

Листинг 1 – `struct pglist_data`

В архитектурах NUMA структуры узлов создаются специфичным для архитектуры кодом на ранних этапах загрузки системы (boot). Обычно эти структуры выделяются локально на банков памяти, который они представляют. В архитектурах UMA используется только одна статическая структура `pg_data_t`, называемая `contig_page_data`.

Всё физическое адресное пространство разделено на один или несколько блоков, называемых зонами (англ. *zones*), которые представляют диапазоны памяти. Эти диапазоны обычно определяются аппаратными ограничениями на доступ к физической памяти. Внутри каждого узла определённая зона памяти описывается с помощью структуры `struct zone`. [1]

1.2.3 Зоны в Linux

Каждый NUMA-узел разделяется на зоны (`ZONE_*`), чтобы учитывать аппаратные ограничения:

- 1) `ZONE_DMA` и `ZONE_DMA32` исторически представляют области памяти, подходящие для DMA-операций (Direct Memory Access) периферийных устройств, которые не могут обращаться ко всей адресуемой памяти;
- 2) `ZONE_NORMAL` предназначена для обычной памяти, к которой ядро всегда имеет доступ;
- 3) `ZONE_HIGHMEM` — это часть физической памяти, которая не отображается постоянно в таблицы страниц ядра. Доступ к памяти в этой зоне возможен только с использованием временных отображений (*temporary mappings*). Эта зона применяется только в некоторых 32-битных архитектурах;
- 4) `ZONE_MOVABLE` — это зона обычной памяти, но её содержимое можно перемещать (для оптимизации фрагментации);
- 5) `ZONE_DEVICE` предназначена для памяти, находящейся на устройствах (например, постоянная память `PMEM` или память `GPU`). Эта память имеет другие характеристики, отличающиеся от обычной оперативной памяти (`RAM`). `ZONE_DEVICE` используется, чтобы предоста-

вить драйверам устройств структуры `struct page` и механизмы управления памятью для работы с физическими адресами, определенными устройством. [1]

1.2.4 Структура `struct page` и массив `mem_map`

В ядре Linux каждая физическая страница памяти представляется структурой `struct page`, содержащей метаданные, необходимые для её управления. Все структуры `struct page` организованы в массив `mem_map`, который создаётся при инициализации системы и позволяет ядру отслеживать и управлять всей физической памятью.

Массив `mem_map` обеспечивает быстрый доступ к структуре `struct page` по номеру страницы (PFN, Page Frame Number).

Далее представлен листинг структуры `struct page`.

```
1 struct page {
2     /* Atomic flags , some possibly updated asynchronously */
3     unsigned long flags;
4     /*
5      * Five words (20/40 bytes) are available in this union.
6      * WARNING: bit 0 of the first word is used for PageTail().
7      * That
8      * means the other users of this union MUST NOT use the bit to
9      * avoid collision and false-positive PageTail().
10    */
11    union {
12        struct { /* Page cache and anonymous pages */
13            /**
14             * @lru: Pageout list , eg. active_list protected by
15             * lruvec->lru_lock. Sometimes used as a generic list
16             * by the page owner.
17             */
18            union {
19                struct list_head lru;
20                /* Or, for the Unevictable "LRU list" slot */
21                struct {
22                    /* Always even , to negate PageTail */
23                    void * __filler;
24                    /* Count page's or folio's mlocks */
25                    unsigned int mlock_count;
```

```

25         };
26         /* Or, free page */
27         struct list_head buddy_list;
28         struct list_head pcp_list;
29     };
30     /* See page-flags.h for PAGE_MAPPING_FLAGS */
31     struct address_space *mapping;
32     union {
33         pgoff_t index;          /* Our offset within mapping.
34                                 */
35         unsigned long share;    /* share count for fsdax
36                                 */
37     };
38     /**
39     * @private: Mapping-private opaque data.
40     * Usually used for buffer_heads if PagePrivate.
41     * Used for swp_entry_t if PageSwapCache.
42     * Indicates order in the buddy system if PageBuddy.
43     */
44     unsigned long private;
45 };
46 ...
47 struct {      /* ZONE_DEVICE pages */
48     /** @pgmap: Points to the hosting device page map. */
49     struct dev_pagemap *pgmap;
50     void *zone_device_data;
51     /*
52      * ZONE_DEVICE private pages are counted as being
53      * mapped so the next 3 words hold the mapping, index,
54      * and private fields from the source anonymous or
55      * page cache page while the page is migrated to
56      * device
57      * private memory.
58      * ZONE_DEVICE MEMORY_DEVICE_FS_DAX pages also
59      * use the mapping, index, and private fields when
60      * pmem backed DAX files are mapped.
61      */
62 };
63
64 /** @rcu_head: You can use this to free a page by RCU. */
65 struct rcu_head rcu_head;

```

```

63     };
64
65     union {          /* This union is 4 bytes in size. */
66         /*
67          * If the page can be mapped to userspace, encodes the
68          * number
69          * of times this page is referenced by a page table.
70          */
71         atomic_t _mapcount;
72         /*
73          * If the page is neither PageSlab nor mappable to
74          * userspace,
75          * the value stored here may help determine what this page
76          * is used for. See page-flags.h for a list of page types
77          * which are currently stored here.
78          */
79         unsigned int page_type;
80     };
81     ...
82     /*
83      * On machines where all RAM is mapped into kernel address
84      * space,
85      * we can simply calculate the virtual address. On machines
86      * with
87      * highmem some memory is mapped into kernel virtual memory
88      * dynamically, so we need a place to store that address.
89      * Note that this field could be 16 bits on x86 ... ;)
90      *
91      * Architectures with slow multiplication can define
92      * WANT_PAGE_VIRTUAL in asm/page.h
93      */
94     #if defined(WANT_PAGE_VIRTUAL)
95         void *virtual;          /* Kernel virtual address (NULL if
96                                 not kmapped, ie. highmem) */
97     #endif /* WANT_PAGE_VIRTUAL */
98     ...
99 };

```

Листинг 2 – struct page

На рисунке ниже представлено отношение между узлами памяти, зонами и страницами.

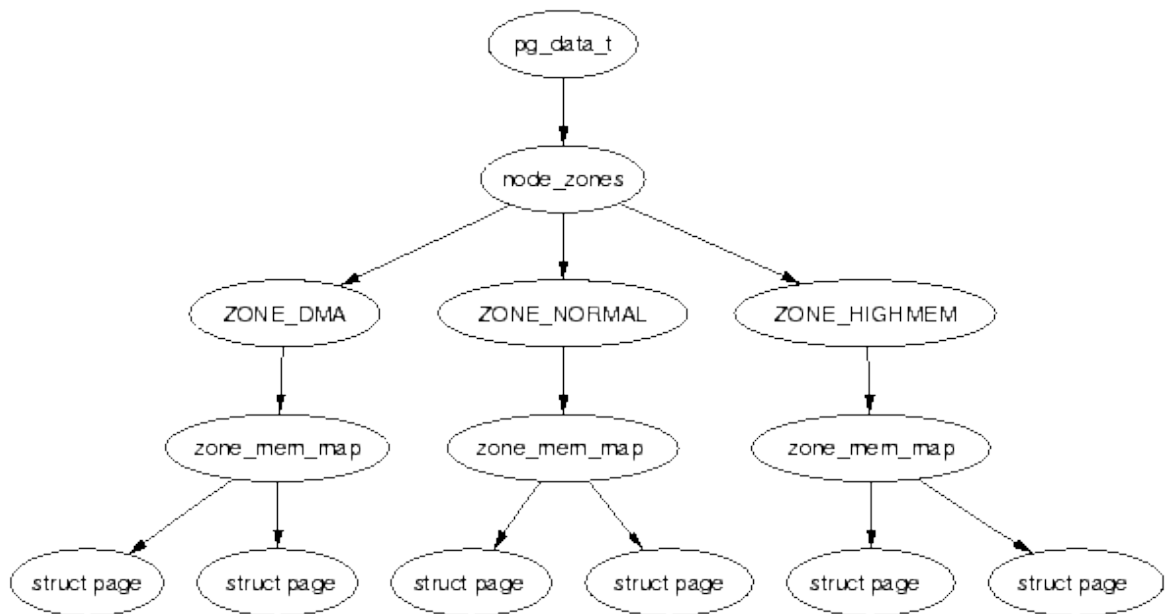


Рисунок 2 – Отношение между узлами памяти, зонами и страницами [2]

1.2.5 SLAB-кэш

Выделение памяти через buddy allocator (используемый для `alloc_pages()`) неэффективно для мелких объектов, поскольку:

- Использование целых страниц для маленьких структур ведет к фрагментации;
- Buddy-аллокатор медленный при работе с большим количеством небольших объектов.

SLAB-аллокатор решает эту проблему, создавая пулы (SLAB-кэши) для повторного использования объектов одного типа.

SLAB-аллокатор — это механизм управления динамической памятью в ядре Linux, предназначенный для эффективного выделения и повторного использования объектов фиксированного размера (например, `task_struct`, `inode`, `dentry`). Он является частью системы управления памятью и предназначен для работы с небольшими объектами, которые часто выделяются и освобождаются.

Работает он следующим образом:

- 1) Ядро создает кэш (`kmem_cache_create()`) для типа объектов (например, `task_struct`);

- 2) SLAB-аллокатор выделяет группу объектов сразу и хранит их в struct slab;
- 3) При kmem_cache_alloc() ядро берет объект из кэша, а не выделяет память заново;
- 4) Когда объект освобождается (kmem_cache_free()), он возвращается в кэш, а не в buddy allocator;
- 5) Если в кэше не осталось свободных объектов, SLAB-аллокатор запрашивает новые страницы через alloc_pages().

1.2.6 Структуры struct kmem_cache и struct slab

Ниже приведены структуры ядра struct kmem_cache и struct slab, используемые при работе со SLAB-аллокатором.

```
1 struct kmem_cache {
2     struct array_cache __percpu *cpu_cache;
3     ...
4     unsigned int size;
5     ...
6     slab_flags_t flags;      /* constant flags */
7     unsigned int num;        /* # of objs per slab */
8     ...
9     const char *name;
10    struct list_head list;
11    int refcount;
12    int object_size;
13    int align;
14    ...
15    struct kmem_cache_node *node[MAX_NUMNODES];
16 };
```

Листинг 3 – struct kmem_cache

```
1 struct slab {
2     unsigned long __page_flags;
3 #if defined(CONFIG_SLAB)
4     struct kmem_cache *slab_cache;
5     union {
6         struct {
```

```

7         struct list_head slab_list;
8         void *freelist; /* array of free object indexes */
9         void *s_mem;    /* first object */
10    };
11    struct rcu_head rcu_head;
12 };
13 unsigned int active;
14 ...
15 #elif defined(CONFIG_SLUB)
16     struct kmem_cache *slab_cache;
17     union {
18         struct {
19             union {
20                 struct list_head slab_list;
21                 ...
22             };
23             /* Double-word boundary */
24             union {
25                 struct {
26                     void *freelist; /* first free object */
27                     union {
28                         unsigned long counters;
29                         struct {
30                             unsigned inuse:16;
31                             unsigned objects:15;
32                             unsigned frozen:1;
33                         };
34                     };
35                 };
36                 ...
37             };
38         };
39         struct rcu_head rcu_head;
40     };
41     ...
42 #endif
43 };

```

Листинг 4 – struct slab

1.3 Функции и системные вызовы

Для работы с памятью в ядре Linux предусмотрены функции

- `alloc_pages`,
- `kmem_cache_create`,
- `kmem_cache_alloc`,
- `kmem_cache_free`,
- `kmem_cache_destroy`,
- `kmalloc`, `kfree`

и другие.

1.3.1 `alloc_pages`

Функция `alloc_pages()` вызывает основную функцию выделения страниц — `__alloc_pages()`, которая выполняет несколько шагов:

- 1) Выбор зоны памяти;
- 2) Поиск подходящего блока в buddy allocator;
- 3) Выделение памяти.

Выбор зоны памяти

Выбор зоны памяти происходит следующим образом — `__alloc_pages()` определяет, из какой зоны (`ZONE_*`) выделять память:

- `ZONE_DMA`;
- `ZONE_NORMAL`;
- `ZONE_HIGHMEM`;
- `ZONE_MOVABLE`;
- `ZONE_DEVICE`;

После чего функция `get_page_from_freelist()` ищет свободные страницы в нужной зоне:

```
1 struct page *page = get_page_from_freelist(gfp_mask, order,
      alloc_flags, alloc_context);
```

Поиск подходящего блока в buddy allocator

Buddy allocator хранит свободные страницы в списках (`free_area`).

- 1) Если есть подходящий свободный блок, он разбивается и выделяется.
- 2) Если нет свободного блока нужного размера, система ищет больше страниц.
- 3) Если памяти недостаточно, включается свопинг (`swap`) или OOM-Killer.

Buddy allocator выделяет память вызовом `remove_from_free_list()`:

```
1 page = remove_from_free_list(zone, order);
```

Причем если

- `order = 0` — выделяется 1 страница;
- `order = 1` — выделяется 2 страницы (2^1);
- `order = 2` — выделяется 4 страницы (2^2)

и так далее.

Выделение памяти

Если найдена свободная страница, ядро

- Помечает её как занятую (`PageReserved`);
- Обновляет структуру `struct page`, устанавливая флаги и счётчик ссылок;
- Возвращает `struct page *`;

После чего функция `prep_new_page()` инициализирует страницу:

```
1 if (page) prep_new_page(page, order, gfp_mask);
```

1.3.2 kmem_cache_*

Далее будут рассмотрены функции `kmem_cache_alloc`, `kmem_cache_free`, `kmem_cache_create`, `kmem_cache_destroy`.

kmem_cache_create

Данная функция предназначена для создания SLAB-кэша.

```
1 struct kmem_cache *kmem_cache_create(const char *name, size_t  
   size, size_t align, slab_flags_t flags, void (*ctor)(void *));
```

Аргументы:

- 1) `name` — Имя кэша (отображается в `/proc/slabinfo`);
- 2) `size` — Размер объекта в байтах;
- 3) `align` — Выравнивание объекта (0 — авто);
- 4) `flags` — Флаги SLAB (`SLAB_HWCACHE_ALIGN`, `SLAB_PANIC`);
- 5) `ctor` — Функция-конструктор (инициализация объекта, может быть `NULL`).

Данная функция

- Выделяет SLAB-кэш для объектов одного типа;
- Создает `struct kmem_cache`, которая управляет списком объектов;
- Создает SLAB-пулы (`struct slab`).

Пример создания кэша для `task_struct`:

```
1 struct kmem_cache *task_struct_cache;  
2 task_struct_cache = kmem_cache_create("task_struct_cache",  
   sizeof(struct task_struct), 0, SLAB_HWCACHE_ALIGN, NULL);
```

kmem_cache_alloc

Данная функция предназначена для выделения объекта из SLAB-кэша.

```
1 void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

Аргументы:

- 1) `cachep` — Указатель на кэш (`struct kmem_cache *`);
- 2) `flags` — Флаги выделения памяти (`GFP_KERNEL`, `GFP_ATOMIC`).

Данная функция

- Ищет свободный объект в `kmem_cache`;
- Если кэш пуст, вызывает `alloc_pages()` для выделения новых страниц;
- Возвращает указатель на объект.

Пример выделения объекта кэша для `task_struct`:

```
1 struct task_struct *task = kmem_cache_alloc(task_struct_cache ,  
      GFP_KERNEL);
```

kmem_cache_free

Данная функция предназначена для освобождения объекта.

```
1 void kmem_cache_free(struct kmem_cache *cachep , void *objp);
```

Аргументы:

- 1) `cachep` — Указатель SLAB-кэш;
- 2) `objp` — Указатель на объект, который нужно освободить.

Данная функция

- Добавляет объект обратно в кэш;
- Если кэш заполнен, объект возвращается в `buddy allocator`;
- Если используется SLAB, объект остаётся в `struct slab`.

Пример освобождения `task_struct`:

```
1 kmem_cache_free(task_struct_cache , task);
```

kmem_cache_destroy

Данная функция предназначена для удаления SLAB-кэша.

```
1 void kmem_cache_destroy(struct kmem_cache *cachep);
```

Аргументы:

- cachep — Указатель на struct kmem_cache;

Данная функция

- Освобождает все объекты в кэше;
- Возвращает все занятые страницы в buddy allocator;
- Удаляет kmem_cache из списка SLAB-кэшей.

Пример удаления кэша:

```
1 kmem_cache_destroy(task_struct_cache);
```

1.3.3 kmalloc

Функция kmalloc() используется для выделения памяти в пространстве ядра и работает через SLAB-аллокатор или buddy allocator, в зависимости от размера запроса и конфигурации системы.

```
1 void *kmalloc(size_t size, gfp_t flags);
```

Аргументы:

- size — Размер выделяемой памяти (в байтах);
- flags — Флаги выделения памяти (GFP_KERNEL, GFP_ATOMIC и т. д.).

Функция kmalloc() может работать через разные механизмы, в зависимости от размера запроса. Так для маленьких объектов ($\text{size} \leq \text{PAGE_SIZE}$) выделение происходит через SLAB, а для больших объектов ($\text{size} > \text{PAGE_SIZE}$) выделение происходит через alloc_pages().

Ниже представлены листинги функций kmalloc, __kmalloc и kmalloc_large.

```

1 void *kmalloc(size_t size , gfp_t flags)
2 {
3     return __kmalloc(size , flags);
4 }

```

Листинг 5 – kmalloc

```

1 void *__kmalloc(size_t size , gfp_t flags)
2 {
3     struct kmem_cache *cache;
4     cache = kmalloc_slab(size , flags);
5     if (unlikely(!cache))
6         return kmalloc_large(size , flags);
7     return kmem_cache_alloc(cache , flags);
8 }

```

Листинг 6 – __kmalloc

```

1 static void *kmalloc_large(size_t size , gfp_t flags)
2 {
3     struct page *page;
4     page = alloc_pages(flags , get_order(size));
5     if (!page)
6         return NULL;
7     return page_address(page);
8 }

```

Листинг 7 – kmalloc_large

Функция `kmalloc_slab()` находит нужный SLAB-кэш (`kmalloc-32`, `kmalloc-64`, `kmalloc-128` и т.д.), после чего, если есть свободные объекты, они выделяются через `kmem_cache_alloc()`.

Примеры использования kmalloc

```

1 void *ptr = kmalloc(64, GFP_KERNEL);
2 if (!ptr) printk(KERN_ERR "kmalloc\n");

```

Листинг 8 – Выделение памяти в SLAB-кэше

```

1 void *ptr = kmalloc(8192, GFP_KERNEL);

```

Листинг 9 – Выделение памяти через `alloc_pages`

1.3.4 kfree

Функция `kfree()` освобождает память. Если память была выделена через SLAB, объект возвращается в SLAB-кэш. Если память была выделена через `alloc_pages()`, страницы освобождаются через `__free_pages()`.

```
1 kfree ( ptr ) ;
```

Листинг 10 – `kfree`

1.4 Анализ способов мониторинга памяти

Мониторинг выделения и использования памяти в ядре Linux может осуществляться разными способами, включая перехват системных вызовов и хуки ядра, а также использование встроенных системных интерфейсов.

1.4.1 Перехват системных вызовов и хуки ядра

Существует множество способов перехвата системных вызовов. Далее будут рассмотрены самые распространенные — `kprobes`, `tracepoints`, `ftrace`, `perf` и `eBPF`.

kprobes

Механизм `kprobes` позволяет динамически устанавливать точки останова в любую функцию ядра и собирать отладочную информацию без нарушения работы системы. С помощью него возможно перехватывать практически любой адрес в коде ядра, указывая обработчик, который будет вызван при срабатывании точки останова. [3]

В настоящее время существует два типа проб (`probes`):

- 1) `kprobes` — стандартные точки перехвата, которые можно вставить практически в любую инструкцию ядра;
- 2) `kretprobes` (`return probes`) — перехватывают момент выхода из указанной функции (при её возврате).

Обычно `kprobes` используется в виде загружаемого модуля ядра. Функция инициализации модуля устанавливает (регистрирует) одну или несколько `kprobe`. Функция выхода удаляет их. Регистрация выполняется с помощью

функции `register_kprobe()`, в которой указывается адрес точки перехвата и обработчик, который должен выполняться при её срабатывании. [3]

Ниже приведена структура `struct kprobe`.

```
1 struct kprobe {
2     struct hlist_node hlist;
3
4     /* list of kprobes for multi-handler support */
5     struct list_head list;
6
7     /*count the number of times this probe was temporarily
8        disarmed */
9     unsigned long nmisses;
10
11    /* location of the probe point */
12    kprobe_opcode_t *addr;
13
14    /* Allow user to indicate symbol name of the probe point */
15    const char *symbol_name;
16
17    /* Offset into the symbol */
18    unsigned int offset;
19
20    /* Called before addr is executed. */
21    kprobe_pre_handler_t pre_handler;
22
23    /* Called after addr is executed, unless... */
24    kprobe_post_handler_t post_handler;
25
26    /* Saved opcode (which has been replaced with breakpoint) */
27    kprobe_opcode_t opcode;
28
29    /* copy of the original instruction */
30    struct arch_specific_insn ainsn;
31
32    /* Indicates various status flags.
33       * Protected by kprobe_mutex after this kprobe is registered.
34       */
35    u32 flags;
36};
```

Листинг 11 – `struct kprobe`

Пример загружаемого модуля ядра, использующего kprobes для перехвата `kmalloc()`:

```
1 #include <linux/kprobes.h>
2 #include <linux/module.h>
3 #include <linux/slab.h>
4
5 static int handler_pre(struct kprobe *p, struct pt_regs *regs) {
6     size_t size = regs->di;
7     printk(KERN_INFO "kmalloc_called:_%zu_bytes\n", size);
8     return 0;
9 }
10
11 static struct kprobe kp = {
12     .symbol_name = "__kmalloc",
13     .pre_handler = handler_pre,
14 };
15
16 static int __init kprobe_init(void) {
17     int ret = register_kprobe(&kp);
18     if (ret < 0) {
19         pr_err("Failed_to_register_kprobe:%d\n", ret);
20         return ret;
21     }
22     pr_info("kprobe_for_kmalloc_installed\n");
23     return 0;
24 }
25
26 static void __exit kprobe_exit(void) {
27     unregister_kprobe(&kp);
28     pr_info("kprobe_removed\n");
29 }
30
31 module_init(kprobe_init);
32 module_exit(kprobe_exit);
```

tracepoints

Точка трассировки (tracepoint), размещённая в коде, предоставляет возможность вызвать функцию (пробу, probe), которую можно назначить во время выполнения. Если tracepoint «включен» (к нему подключена probe), то

при его срабатывании вызывается соответствующая функция. Если `tracpoint` «выключен» (к нему не подключено обработчиков), он не влияет на выполнение кода, за исключением небольшой временной задержки и небольших затрат памяти. Функция-проба вызывается каждый раз при выполнении `tracpoint`. Выполняется в том же контексте, что и вызывающая функция. После завершения работы обработчика выполнение возвращается в исходное место, продолжая выполнение основной программы.

Точки трассировки используются для трассировки и анализа производительности. Их можно вставлять в критически важные участки кода, чтобы отслеживать работу системы. [4]

ftrace

Ftrace — это фреймворк, состоящий из нескольких различных утилит трассировки. Одно из самых распространенных применений `ftrace` — трассировка событий. По всему ядру расположены сотни статических точек событий, которые можно включить через файловую систему `tracefs`, чтобы посмотреть, что происходит в определенных частях ядра. [5]

В листинге 23 приложения А приведен пример реализации загружаемого модуля ядра для мониторинга системных вызовов `__kmalloc` и `kmem_cache_alloc` с использованием фреймворка `ftrace`.

1.4.2 Сравнительный анализ подходов

1.4.3 Выбор метода мониторинга

2 Конструкторский раздел

3 Технологический раздел

3.1 Средства реализации

Для реализации ПО был выбран язык программирования C [6], поскольку в нем есть все инструменты для реализации загружаемого модуля ядра.

В качестве среды разработки был выбран Neovim [7], так как он позволяет редактировать файлы с исходным кодом программы.

3.2 Реализация загружаемого модуля ядра

Ниже представлены реализации функций и структур загружаемого модуля ядра.

```
1 struct mem_alloc_entry {
2     pid_t pid;
3     char func_name[FUNC_NAME_LEN];
4     char comm[TASK_COMM_LEN];
5     char cache_name[CACHE_NAME_LEN];
6     char is_cache;
7     size_t size;
8     struct list_head list;
9 };
```

Листинг 12 – Структура struct mem_alloc_entry для логирования информации

```
1 static int pre_kmalloc_handler(struct kprobe *p, struct pt_regs
   *regs)
2 {
3     struct task_struct *task = current;
4     size_t size = regs->di;
5     struct mem_alloc_entry *entry;
6
7     if (size == 0)
8         return 0;
9
10    entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
11    if (!entry)
```

```

12         return 0;
13
14     strncpy(entry->func_name, "kmalloc", FUNC_NAME_LEN);
15     entry->pid = task->pid;
16     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
17     entry->size = size;
18     entry->is_cache = 0;
19
20     spin_lock(&mem_lock);
21     prune_oldest_entry();
22     list_add_tail(&entry->list, &mem_alloc_list);
23     entry_count++;
24     spin_unlock(&mem_lock);
25
26     return 0;
27 }

```

Листинг 13 – Функция pre_kmalloc_handler

```

1 static int pre_kmem_cache_alloc_handler(struct kprobe *p, struct
    pt_regs *regs)
2 {
3     void *cachep = (void *)regs->di;
4     const char **name_ptr;
5     const char *cache_name;
6     struct task_struct *task = current;
7
8     if (!cachep)
9         return 0;
10
11     name_ptr = (const char **)((char *)cachep + 96);
12     if (!name_ptr)
13         return 0;
14
15     cache_name = *name_ptr;
16     if (!cache_name)
17         return 0;
18
19     size_t cache_size = (size_t)((char *)cachep + 24));
20     if (!cache_size)
21         return 0;
22

```

```

23     struct mem_alloc_entry *entry;
24
25     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
26     if (!entry)
27         return 0;
28
29     strncpy(entry->func_name, "kmem_cache_alloc", FUNC_NAME_LEN);
30     entry->pid = task->pid;
31     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
32     strncpy(entry->cache_name, cache_name, CACHE_NAME_LEN);
33     entry->size = cache_size;
34     entry->is_cache = 1;
35
36     spin_lock(&mem_lock);
37     prune_oldest_entry();
38     list_add_tail(&entry->list, &mem_alloc_list);
39     entry_count++;
40     spin_unlock(&mem_lock);
41
42     return 0;
43 }

```

Листинг 14 – Функция pre_kmem_cache_alloc_handler

```

1 static struct kprobe kp_kmalloc = {
2     .symbol_name = "__kmalloc",
3     .pre_handler = pre_kmalloc_handler,
4 };

```

Листинг 15 – Проба kp_kmalloc

```

1 static struct kprobe kp_kmem_cache_alloc = {
2     .symbol_name = "kmem_cache_alloc",
3     .pre_handler = pre_kmem_cache_alloc_handler,
4 };

```

Листинг 16 – Проба kp_kmalloc

```

1 static int __init mem_monitor_init(void)
2 {
3     int ret;
4
5     ret = register_kprobe(&kp_kmalloc);
6     if (ret < 0)

```

```

7         goto kmalloc_error;
8
9     ret = register_kprobe(&kp_kmalloc_large);
10    if (ret < 0)
11        goto kmalloc_large_error;
12
13    ret = register_kprobe(&kp_kmalloc_slab);
14    if (ret < 0)
15        goto kmalloc_slab_error;
16
17    ret = register_kprobe(&kp_kmalloc_node);
18    if (ret < 0)
19        goto kmalloc_node_error;
20
21    ret = register_kprobe(&kp_kmem_cache_alloc);
22    if (ret < 0)
23        goto kmem_cache_alloc_error;
24
25    goto ok;
26
27 kmem_cache_alloc_error:
28     unregister_kprobe(&kp_kmalloc_node);
29 kmalloc_node_error:
30     unregister_kprobe(&kp_kmalloc_slab);
31 kmalloc_slab_error:
32     unregister_kprobe(&kp_kmalloc_large);
33 kmalloc_large_error:
34     unregister_kprobe(&kp_kmalloc);
35 kmalloc_error:
36     return ret;
37
38 ok:
39     proc_create(PROC_FILENAME, 0444, NULL, &mem_proc_ops);
40     pr_info("Memory_monitor_module_loaded.\n");
41     return 0;
42 }

```

Листинг 17 – Функция init загружаемого модуля

```

1 static void __exit mem_monitor_exit(void)
2 {
3     struct mem_alloc_entry *entry, *tmp;

```

```

4
5     unregister_kprobe(&kp_kmalloc);
6     unregister_kprobe(&kp_kmalloc_large);
7     unregister_kprobe(&kp_kmalloc_slab);
8     unregister_kprobe(&kp_kmalloc_node);
9     unregister_kprobe(&kp_kmem_cache_alloc);
10    remove_proc_entry(PROC_FILENAME, NULL);
11
12    spin_lock(&mem_lock);
13    list_for_each_entry_safe(entry, tmp, &mem_alloc_list, list) {
14        list_del(&entry->list);
15        kfree(entry);
16    }
17    spin_unlock(&mem_lock);
18
19    pr_info("Memory_monitor_module_unloaded.\n");
20 }

```

Листинг 18 – Функция exit загружаемого модуля

3.3 Реализация Makefile

Ниже представлен Makefile для сборки загружаемого модуля ядра.

```

1 obj-m := mem_monitor.o
2 KDIR := /lib/modules/$(shell uname -r)/build
3 PWD := $(shell pwd)
4
5 all:
6     $(MAKE) -C $(KDIR) M=$(PWD) modules
7
8 clean:
9     $(MAKE) -C $(KDIR) M=$(PWD) clean

```

Листинг 19 – Makefile

3.4 Реализация программы для генерации потоков

Ниже представлена программа для генерации потоков, используемая для мониторинга выделения памяти потоками.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 void* thread_func(void* arg)
7 {
8     void *a = malloc(8192);
9     free(a);
10    return NULL;
11 }
12
13 int main()
14 {
15     int i;
16     const int num_threads = 1000;
17     pthread_t threads[num_threads];
18
19     for (i = 0; i < num_threads; i++)
20         pthread_create(&threads[i], NULL, thread_func, NULL);
21
22     for (i = 0; i < num_threads; i++)
23         pthread_join(threads[i], NULL);
24
25     return 0;
```

Листинг 20 – threads.c

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристики устройства, на котором запускалась программа, представлены ниже.

- 1) Процессор: AMD Ryzen 7 4700U 2.0 ГГц [8], 8 физических ядер, 8 потоков;
- 2) Оперативная память: 8 ГБ, DDR4, 3200 МГц;
- 3) Операционная система: Arch Linux [9];
- 4) Версия ядра: 6.13.3.

Технические характеристики виртуальной машины QEMU [10], на которой запускалась программа, представлены ниже.

- 1) Виртуальная машина: qemu-system-x86_64;
- 2) Оперативная память: 2 ГБ;
- 3) Операционная система: Arch Linux;
- 4) Версия ядра: 6.6 (собрано вручную с флагом CONFIG_KALLSYMS=y и другими для возможности более глубокой отладки ядра).

4.2 Демонстрация работы ПО

```
1 2 3 4 5 6 7 8 2025-02-21 Friday 12:32:05 Volume muted | Battery 49 | Qwerty
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 64 bytes in cache 'kmalloc-64'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 160 bytes in cache 'p9_req_t'
[kmalloc]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 72 bytes in cache 'radix_tree_node'
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 160 bytes in cache 'p9_req_t'
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 72 bytes in cache 'radix_tree_node'
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 160 bytes in cache 'p9_req_t'
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 4096 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 72 bytes in cache 'radix_tree_node'
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 80 bytes in cache 'Acpi-State'
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 104 bytes in cache 'buffer_head'
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 104 bytes in cache 'buffer_head'
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 192 bytes in cache 'bio-184'
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 192 bytes in cache 'bio-184'
[kmalloc_slab]: Process 'jbd2/vda-8' with PID 99 allocated 64 bytes
[kmalloc_slab]: Process 'jbd2/vda-8' with PID 99 allocated 64 bytes
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 104 bytes in cache 'buffer_head'
[kmem_cache_alloc]: Process 'jbd2/vda-8' with PID 99 allocated 192 bytes in cache 'bio-184'
[kmalloc_slab]: Process 'kworker/3:1H' with PID 129 allocated 32 bytes
[kmalloc_slab]: Process 'kworker/3:1H' with PID 129 allocated 32 bytes
```

Рисунок 3 – Демонстрация работы ПО 1

```
1 2 3 4 5 6 7 8 2025-02-21 Friday 12:32:15 Volume muted | Battery 49 | Qwerty
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 128 bytes in cache 'kmalloc-128'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 64 bytes in cache 'kmalloc-64'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 104 bytes in cache 'anon_vma'
[kmalloc_node]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 128 bytes in cache 'kmalloc-128'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 64 bytes in cache 'kmalloc-64'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 104 bytes in cache 'anon_vma'
[kmalloc_node]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 128 bytes in cache 'kmalloc-128'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 72 bytes in cache 'radix_tree_node'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 168 bytes in cache 'vm_area_struct'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 40 bytes in cache 'vma_lock'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 64 bytes in cache 'kmalloc-64'
[kmem_cache_alloc]: Process 'a.out' with PID 316 allocated 104 bytes in cache 'anon_vma'
[kmalloc_node]: Process 'a.out' with PID 316 allocated 32 bytes
[kmalloc_slab]: Process 'a.out' with PID 316 allocated 32 bytes
```

Рисунок 4 – Демонстрация работы ПО 2

1	2	3	4	5	6	7	8	2025-02-21 Friday 12:34:11							Volume muted Battery 48 Qwerty		
dma-kmalloc-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0		
dma-kmalloc-96	0	0	96	42	1	:	tunables	0	0	0	:	slabdata	0	0	0		
dma-kmalloc-64	0	0	64	64	1	:	tunables	0	0	0	:	slabdata	0	0	0		
dma-kmalloc-32	0	0	32	128	1	:	tunables	0	0	0	:	slabdata	0	0	0		
dma-kmalloc-16	0	0	16	256	1	:	tunables	0	0	0	:	slabdata	0	0	0		
dma-kmalloc-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-8k	0	0	8192	4	8	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-4k	0	0	4096	8	8	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-2k	0	0	2048	16	8	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-1k	0	0	1024	32	8	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-512	0	0	512	32	4	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-256	96	96	256	32	2	:	tunables	0	0	0	:	slabdata	3	3	0		
kmalloc-rcl-192	7938	7938	192	21	1	:	tunables	0	0	0	:	slabdata	378	378	0		
kmalloc-rcl-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-rcl-96	294	294	96	42	1	:	tunables	0	0	0	:	slabdata	7	7	0		
kmalloc-rcl-64	576	576	64	64	1	:	tunables	0	0	0	:	slabdata	9	9	0		
kmalloc-rcl-32	256	256	32	128	1	:	tunables	0	0	0	:	slabdata	2	2	0		
kmalloc-rcl-16	256	256	16	256	1	:	tunables	0	0	0	:	slabdata	1	1	0		
kmalloc-rcl-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0		
kmalloc-8k	28	28	8192	4	8	:	tunables	0	0	0	:	slabdata	7	7	0		
kmalloc-4k	352	352	4096	8	8	:	tunables	0	0	0	:	slabdata	44	44	0		
kmalloc-2k	432	432	2048	16	8	:	tunables	0	0	0	:	slabdata	27	27	0		
kmalloc-1k	704	704	1024	32	8	:	tunables	0	0	0	:	slabdata	22	22	0		
kmalloc-512	1248	1248	512	32	4	:	tunables	0	0	0	:	slabdata	39	39	0		
kmalloc-256	2470	3360	256	32	2	:	tunables	0	0	0	:	slabdata	105	105	0		
kmalloc-192	1470	1470	192	21	1	:	tunables	0	0	0	:	slabdata	70	70	0		
kmalloc-128	15168	15168	128	32	1	:	tunables	0	0	0	:	slabdata	474	474	0		
kmalloc-96	15802	15918	96	42	1	:	tunables	0	0	0	:	slabdata	379	379	0		
kmalloc-64	5534	5632	64	64	1	:	tunables	0	0	0	:	slabdata	88	88	0		
kmalloc-32	4217	4480	32	128	1	:	tunables	0	0	0	:	slabdata	35	35	0		
kmalloc-16	7168	7168	16	256	1	:	tunables	0	0	0	:	slabdata	28	28	0		
kmalloc-8	16896	16896	8	512	1	:	tunables	0	0	0	:	slabdata	33	33	0		
kmem_cache_node	320	320	64	64	1	:	tunables	0	0	0	:	slabdata	5	5	0		
kmem_cache	224	224	256	32	2	:	tunables	0	0	0	:	slabdata	7	7	0		
root@archlinux: ~																	
# [7.588178][T193] (udev-worker) (193) used greatest stack depth: 12672 bytes left																	
]																	

Рисунок 7 – Вывод cat /proc/slabinfo до запуска программы для генерации ПОТОКОВ

1	2	3	4	5	6	7	8	2025-02-21 Friday 12:38:21						Volume muted Battery 47 Qwerty		
dma-kmalloc-256	0	0	256	32	2	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-192	0	0	192	21	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-96	0	0	96	42	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-64	0	0	64	64	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-32	0	0	32	128	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-16	0	0	16	256	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-8k	0	0	8192	4	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-4k	0	0	4096	8	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-2k	0	0	2048	16	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-1k	0	0	1024	32	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-512	0	0	512	32	4	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-256	160	160	256	32	2	:	tunables	0	0	0	:	slabdata	5	5	0	
kmalloc-rcl-192	7869	7917	192	21	1	:	tunables	0	0	0	:	slabdata	377	377	0	
kmalloc-rcl-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-96	294	294	96	42	1	:	tunables	0	0	0	:	slabdata	7	7	0	
kmalloc-rcl-64	576	576	64	64	1	:	tunables	0	0	0	:	slabdata	9	9	0	
kmalloc-rcl-32	256	256	32	128	1	:	tunables	0	0	0	:	slabdata	2	2	0	
kmalloc-rcl-16	256	256	16	256	1	:	tunables	0	0	0	:	slabdata	1	1	0	
kmalloc-rcl-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-8k	32	32	8192	4	8	:	tunables	0	0	0	:	slabdata	8	8	0	
kmalloc-4k	2320	2320	4096	8	8	:	tunables	0	0	0	:	slabdata	290	290	0	
kmalloc-2k	448	448	2048	16	8	:	tunables	0	0	0	:	slabdata	28	28	0	
kmalloc-1k	704	704	1024	32	8	:	tunables	0	0	0	:	slabdata	22	22	0	
kmalloc-512	11840	11840	512	32	4	:	tunables	0	0	0	:	slabdata	370	370	0	
kmalloc-256	7456	7456	256	32	2	:	tunables	0	0	0	:	slabdata	233	233	0	
kmalloc-192	1512	1512	192	21	1	:	tunables	0	0	0	:	slabdata	72	72	0	
kmalloc-128	17696	17696	128	32	1	:	tunables	0	0	0	:	slabdata	553	553	0	
kmalloc-96	15801	15918	96	42	1	:	tunables	0	0	0	:	slabdata	379	379	0	
kmalloc-64	13760	13760	64	64	1	:	tunables	0	0	0	:	slabdata	215	215	0	
kmalloc-32	8704	8704	32	128	1	:	tunables	0	0	0	:	slabdata	68	68	0	
kmalloc-16	7168	7168	16	256	1	:	tunables	0	0	0	:	slabdata	28	28	0	
kmalloc-8	16896	16896	8	512	1	:	tunables	0	0	0	:	slabdata	33	33	0	
kmem_cache_node	320	320	64	64	1	:	tunables	0	0	0	:	slabdata	5	5	0	
kmem_cache	224	224	256	32	2	:	tunables	0	0	0	:	slabdata	7	7	0	
[user@archlinux v2]\$																

Рисунок 8 – Вывод cat /proc/slabinfo во время запуска программы для генерации потоков

1	2	3	4	5	6	7	8	2025-02-21 Friday 12:45:23						Volume muted Battery 44 Qwerty		
dma-kmalloc-256	0	0	256	32	2	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-192	0	0	192	21	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-96	0	0	96	42	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-64	0	0	64	64	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-32	0	0	32	128	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-16	0	0	16	256	1	:	tunables	0	0	0	:	slabdata	0	0	0	
dma-kmalloc-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-8k	0	0	8192	4	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-4k	0	0	4096	8	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-2k	0	0	2048	16	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-1k	0	0	1024	32	8	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-512	0	0	512	32	4	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-256	192	192	256	32	2	:	tunables	0	0	0	:	slabdata	6	6	0	
kmalloc-rcl-192	7867	7917	192	21	1	:	tunables	0	0	0	:	slabdata	377	377	0	
kmalloc-rcl-128	0	0	128	32	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-rcl-96	294	294	96	42	1	:	tunables	0	0	0	:	slabdata	7	7	0	
kmalloc-rcl-64	576	576	64	64	1	:	tunables	0	0	0	:	slabdata	9	9	0	
kmalloc-rcl-32	256	256	32	128	1	:	tunables	0	0	0	:	slabdata	2	2	0	
kmalloc-rcl-16	256	256	16	256	1	:	tunables	0	0	0	:	slabdata	1	1	0	
kmalloc-rcl-8	0	0	8	512	1	:	tunables	0	0	0	:	slabdata	0	0	0	
kmalloc-8k	32	32	8192	4	8	:	tunables	0	0	0	:	slabdata	8	8	0	
kmalloc-4k	369	464	4096	8	8	:	tunables	0	0	0	:	slabdata	58	58	0	
kmalloc-2k	448	448	2048	16	8	:	tunables	0	0	0	:	slabdata	28	28	0	
kmalloc-1k	704	704	1024	32	8	:	tunables	0	0	0	:	slabdata	22	22	0	
kmalloc-512	11412	11520	512	32	4	:	tunables	0	0	0	:	slabdata	360	360	0	
kmalloc-256	2526	3584	256	32	2	:	tunables	0	0	0	:	slabdata	112	112	0	
kmalloc-192	1512	1512	192	21	1	:	tunables	0	0	0	:	slabdata	72	72	0	
kmalloc-128	15475	15552	128	32	1	:	tunables	0	0	0	:	slabdata	486	486	0	
kmalloc-96	15801	15918	96	42	1	:	tunables	0	0	0	:	slabdata	379	379	0	
kmalloc-64	4721	5504	64	64	1	:	tunables	0	0	0	:	slabdata	86	86	0	
kmalloc-32	4949	5248	32	128	1	:	tunables	0	0	0	:	slabdata	41	41	0	
kmalloc-16	7152	7168	16	256	1	:	tunables	0	0	0	:	slabdata	28	28	0	
kmalloc-8	16896	16896	8	512	1	:	tunables	0	0	0	:	slabdata	33	33	0	
kmem_cache_node	320	320	64	64	1	:	tunables	0	0	0	:	slabdata	5	5	0	
kmem_cache	224	224	256	32	2	:	tunables	0	0	0	:	slabdata	7	7	0	
[user@archlinux v2]\$																

Рисунок 9 – Вывод `cat /proc/slabinfo` после запуска программы для генерации потоков

4.3 Анализ результатов работы ПО

В таблице ниже представлено количество объектов в кэшах kmalloc-256, kmalloc-128 и kmalloc-64 соответственно до, во время и после запуска программы для генерации потоков.

Таблица 1 – Количество объектов в кэше

Кэш	До	Во время	После
kmalloc-256	3360	7456	3584
kmalloc-128	15168	17696	15552
kmalloc-64	5632	13760	5504

Анализируя количество объектов в кэшах kmalloc-256, kmalloc-128 и kmalloc-64 до, во время и после запуска программы для генерации потоков, можно сделать вывод, что во всех трех кэшах наблюдается значительное увеличение количества объектов в период выполнения программы:

- Наибольший рост зафиксирован в kmalloc-64 (на 144.3%), что указывает на активное выделение небольших блоков памяти;
- В kmalloc-256 рост составил 121.9%, что свидетельствует о значительном выделении памяти для относительно крупных объектов;
- В kmalloc-128 рост был менее выраженным (16.7%), но также значительным.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация ядра Linux. Управление физической памятью [Электронный ресурс]. – URL: https://www.kernel.org/doc/html/latest/mm/physical_memory.html (дата обращения: 19.02.2025).
2. Документация ядра Linux. Описание физической памяти [Электронный ресурс]. – URL: <https://www.kernel.org/doc/gorman/html/understand/understand005.html> (дата обращения: 19.02.2025).
3. Документация ядра Linux. Пробы ядра (Kprobes) [Электронный ресурс]. – URL: <https://www.kernel.org/doc/html/latest/trace/kprobes.html> (дата обращения: 19.02.2025).
4. Документация ядра Linux. Использование tracepoints [Электронный ресурс]. – URL: <https://www.kernel.org/doc/html/latest/trace/tracepoints.html> (дата обращения: 19.02.2025).
5. Документация ядра Linux. Трассировщик ftrace [Электронный ресурс]. – URL: <https://www.kernel.org/doc/html/latest/trace/ftrace.html> (дата обращения: 19.02.2025).
6. Документация языка программирования C [Электронный ресурс]. – URL: <https://learn.microsoft.com/en-us/cpp/c-language/?view=msvc-170> (дата обращения: 21.02.2025).
7. Neovim [Электронный ресурс]. – URL: <https://neovim.io> (дата обращения: 21.02.2025).
8. AMD Ryzen 7 4700U [Электронный ресурс]. – URL: <https://www.amd.com/en/product/9096> (дата обращения: 26.01.2025).
9. Arch Linux — A simple, lightweight distribution [Электронный ресурс]. – URL: <https://archlinux.org/> (дата обращения: 21.02.2025).

10. QEMU — A generic and open source machine emulator and virtualizer [Электронный ресурс]. — URL: <https://qemu.org/> (дата обращения: 21.02.2025).

ПРИЛОЖЕНИЕ А

```
1 Руководство пользователя.
2
3 1. Склонировать репозиторий Linux:
4 $ git clone
   https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
5
6 2. Перейти в директорию linux:
7 $ cd linux
8
9 3. Перейти на ветку версии ядра 6.6:
10 $ git checkout v6.6
11
12 4. Зайти в конфигуратор ядра:
13 $ make menuconfig
14
15 5. Включить флаг KALLSYMS:
16 -> General setup
17     -> Configure standard kernel features (expert users) (EXPERT
        [=y])
18     -> Load all symbols for debugging/ksymoops (KALLSYMS [=y])
19
20 6. Сохранить конфигурацию:
21 <Save>
22
23 7. Собрать сжатый образ ядра bzImage:
24 $ make -j {NUM_THREADS} bzImage modules
25
26 8. Склонировать репозиторий для быстрого запуска виртуальной машин
   ы QEMU на основе собранного ядра:
27 $ git clone https://github.com/bgmerrell/vkerndev
28
29 9. Перейти в директорию vkerndev:
30 $ cd vkerndev
31
32 10*. Запустить скрипт для создания виртуальной машины:
33 $ python make_vm.py
34
35 11*. Запустить скрипт для запуска виртуальной машины:
36 $ python run_vm.py
```

```
37
38 12. Поместить программу загружаемого модуля в директорию, указанну
    ю как shared при запуске виртуальной машины:
39
40 13. Собрать программу:
41 $ su user
42 $ cd
43 $ cd host
44 $ cd <Shared папка с программой>
45 $ make
46
47 14. Инициализировать загружаемый модуль:
48 $ sudo insmod mem_monitor.ko
49
50 15. Просмотр информации, создаваемой загружаемым модулем:
51 $ sudo cat /proc/mem_monitor
52
53 16. Выгрузить загружаемый модуль:
54 $ sudo rmmod mem_monitor
55
56 * В скриптах предварительно нужно указать свои настройки и пути к
    файлам, в том числе, к файлу собранного ядра.
```

Листинг 21 – Руководство пользователя

```

1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/kprobes.h>
4 #include <linux/seq_file.h>
5 #include <linux/proc_fs.h>
6 #include <linux/slab.h>
7 #include <linux/sched.h>
8 #include <linux/string.h>
9 #include <linux/spinlock.h>
10
11 MODULE_LICENSE("GPL");
12 MODULE_AUTHOR("Runov_Konstantin");
13 MODULE_DESCRIPTION("Memory_Allocation_Monitoring_Module");
14
15 #define PROC_FILENAME "mem_monitor"
16 #define FUNC_NAME_LEN 128
17 #define CACHE_NAME_LEN 128
18 #define MAX_ENTRIES 10000
19
20 static DEFINE_SPINLOCK(mem_lock);
21 static int entry_count = 0;
22
23 struct mem_alloc_entry {
24     pid_t pid;
25     char func_name[FUNC_NAME_LEN];
26     char comm[TASK_COMM_LEN];
27     char cache_name[CACHE_NAME_LEN];
28     char is_cache;
29     size_t size;
30     struct list_head list;
31 };
32
33 static LIST_HEAD(mem_alloc_list);
34
35 static void prune_oldest_entry(void)
36 {
37     struct mem_alloc_entry *oldest;
38
39     if (entry_count < MAX_ENTRIES)
40         return;
41

```

```

42     oldest = list_first_entry(&mem_alloc_list, struct
        mem_alloc_entry, list);
43     list_del(&oldest->list);
44     kfree(oldest);
45     entry_count--;
46 }
47
48 static int pre_kmalloc_handler(struct kprobe *p, struct pt_regs
        *regs)
49 {
50     struct task_struct *task = current;
51     size_t size = regs->di;
52     struct mem_alloc_entry *entry;
53
54     if (size == 0)
55         return 0;
56
57     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
58     if (!entry)
59         return 0;
60
61     strncpy(entry->func_name, "kmalloc", FUNC_NAME_LEN);
62     entry->pid = task->pid;
63     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
64     entry->size = size;
65     entry->is_cache = 0;
66
67     spin_lock(&mem_lock);
68     prune_oldest_entry();
69     list_add_tail(&entry->list, &mem_alloc_list);
70     entry_count++;
71     spin_unlock(&mem_lock);
72
73     return 0;
74 }
75
76 static int pre_kmalloc_large_handler(struct kprobe *p, struct
        pt_regs *regs)
77 {
78     struct task_struct *task = current;
79     size_t size = regs->di;

```

```

80     struct mem_alloc_entry *entry;
81
82     if (size == 0)
83         return 0;
84
85     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
86     if (!entry)
87         return 0;
88
89     strncpy(entry->func_name, "kmalloc_large", FUNC_NAME_LEN);
90     entry->pid = task->pid;
91     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
92     entry->size = size;
93     entry->is_cache = 0;
94
95     spin_lock(&mem_lock);
96     prune_oldest_entry();
97     list_add_tail(&entry->list, &mem_alloc_list);
98     entry_count++;
99     spin_unlock(&mem_lock);
100
101     return 0;
102 }
103
104 static int pre_kmalloc_slab_handler(struct kprobe *p, struct
    pt_regs *regs)
105 {
106     struct task_struct *task = current;
107     size_t size = regs->di;
108     struct mem_alloc_entry *entry;
109
110     if (size == 0)
111         return 0;
112
113     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
114     if (!entry)
115         return 0;
116
117     strncpy(entry->func_name, "kmalloc_slab", FUNC_NAME_LEN);
118     entry->pid = task->pid;
119     strncpy(entry->comm, task->comm, TASK_COMM_LEN);

```

```

120     entry->size = size;
121     entry->is_cache = 0;
122
123     spin_lock(&mem_lock);
124     prune_oldest_entry();
125     list_add_tail(&entry->list, &mem_alloc_list);
126     entry_count++;
127     spin_unlock(&mem_lock);
128
129     return 0;
130 }
131
132 static int pre_kmalloc_node_handler(struct kprobe *p, struct
    pt_regs *regs)
133 {
134     struct task_struct *task = current;
135     size_t size = regs->di;
136     struct mem_alloc_entry *entry;
137
138     if (size == 0)
139         return 0;
140
141     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
142     if (!entry)
143         return 0;
144
145     strncpy(entry->func_name, "kmalloc_node", FUNC_NAME_LEN);
146     entry->pid = task->pid;
147     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
148     entry->size = size;
149     entry->is_cache = 0;
150
151     spin_lock(&mem_lock);
152     prune_oldest_entry();
153     list_add_tail(&entry->list, &mem_alloc_list);
154     entry_count++;
155     spin_unlock(&mem_lock);
156
157     return 0;
158 }
159

```



```

160 static int pre_kmem_cache_alloc_handler(struct kprobe *p, struct
    pt_regs *regs)
161 {
162     void *cachep = (void *)regs->di;
163     const char **name_ptr;
164     const char *cache_name;
165     struct task_struct *task = current;
166
167     if (!cachep)
168         return 0;
169
170     name_ptr = (const char **)((char *)cachep + 96);
171     if (!name_ptr)
172         return 0;
173
174     cache_name = *name_ptr;
175     if (!cache_name)
176         return 0;
177
178     size_t cache_size = (size_t)((char *)cachep + 24));
179     if (!cache_size)
180         return 0;
181
182     struct mem_alloc_entry *entry;
183
184     entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
185     if (!entry)
186         return 0;
187
188     strncpy(entry->func_name, "kmem_cache_alloc", FUNC_NAME_LEN);
189     entry->pid = task->pid;
190     strncpy(entry->comm, task->comm, TASK_COMM_LEN);
191     strncpy(entry->cache_name, cache_name, CACHE_NAME_LEN);
192     entry->size = cache_size;
193     entry->is_cache = 1;
194
195     spin_lock(&mem_lock);
196     prune_oldest_entry();
197     list_add_tail(&entry->list, &mem_alloc_list);
198     entry_count++;
199     spin_unlock(&mem_lock);

```

```

200
201     return 0;
202 }
203
204 static struct kprobe kp_kmalloc = {
205     .symbol_name = "__kmalloc",
206     .pre_handler = pre_kmalloc_handler,
207 };
208
209 static struct kprobe kp_kmalloc_large = {
210     .symbol_name = "kmalloc_large",
211     .pre_handler = pre_kmalloc_large_handler,
212 };
213
214 static struct kprobe kp_kmalloc_slab = {
215     .symbol_name = "kmalloc_slab",
216     .pre_handler = pre_kmalloc_slab_handler,
217 };
218
219 static struct kprobe kp_kmalloc_node = {
220     .symbol_name = "__kmalloc_node",
221     .pre_handler = pre_kmalloc_node_handler,
222 };
223
224 static struct kprobe kp_kmem_cache_alloc = {
225     .symbol_name = "kmem_cache_alloc",
226     .pre_handler = pre_kmem_cache_alloc_handler,
227 };
228
229 static int mem_proc_show(struct seq_file *m, void *v)
230 {
231     struct mem_alloc_entry *entry;
232
233     spin_lock(&mem_lock);
234     list_for_each_entry(entry, &mem_alloc_list, list) {
235         if (entry->is_cache) {
236             seq_printf(m, "[%s]:_Process_'%s' _with_PID_%d_
237                        allocated_%zu_bytes_in_cache_'%s'\n",
238                        entry->func_name, entry->comm, entry->pid,
239                        entry->size, entry->cache_name);
240         } else {

```

```

239         seq_printf(m, "[%s]:_Process_'%s' _with_PID_%d_
           allocated_%zu_bytes\n",
240                   entry->func_name, entry->comm, entry->pid,
                   entry->size);
241     }
242 }
243 spin_unlock(&mem_lock);
244
245     return 0;
246 }
247
248 static int mem_proc_open(struct inode *inode, struct file *file)
249 {
250     return single_open(file, mem_proc_show, NULL);
251 }
252
253 static const struct proc_ops mem_proc_ops = {
254     .proc_open = mem_proc_open,
255     .proc_read = seq_read,
256     .proc_lseek = seq_lseek,
257     .proc_release = single_release,
258 };
259
260 static int __init mem_monitor_init(void)
261 {
262     int ret;
263
264     ret = register_kprobe(&kp_kmalloc);
265     if (ret < 0)
266         goto kmalloc_error;
267
268     ret = register_kprobe(&kp_kmalloc_large);
269     if (ret < 0)
270         goto kmalloc_large_error;
271
272     ret = register_kprobe(&kp_kmalloc_slab);
273     if (ret < 0)
274         goto kmalloc_slab_error;
275
276     ret = register_kprobe(&kp_kmalloc_node);
277     if (ret < 0)

```

```

278         goto kmalloc_node_error;
279
280     ret = register_kprobe(&kp_kmem_cache_alloc);
281     if (ret < 0)
282         goto kmem_cache_alloc_error;
283
284     goto ok;
285
286 kmem_cache_alloc_error:
287     unregister_kprobe(&kp_kmalloc_node);
288 kmalloc_node_error:
289     unregister_kprobe(&kp_kmalloc_slab);
290 kmalloc_slab_error:
291     unregister_kprobe(&kp_kmalloc_large);
292 kmalloc_large_error:
293     unregister_kprobe(&kp_kmalloc);
294 kmalloc_error:
295     return ret;
296
297 ok:
298     proc_create(PROC_FILENAME, 0444, NULL, &mem_proc_ops);
299     pr_info("Memory_monitor_module_loaded.\n");
300     return 0;
301 }
302
303 static void __exit mem_monitor_exit(void)
304 {
305     struct mem_alloc_entry *entry, *tmp;
306
307     unregister_kprobe(&kp_kmalloc);
308     unregister_kprobe(&kp_kmalloc_large);
309     unregister_kprobe(&kp_kmalloc_slab);
310     unregister_kprobe(&kp_kmalloc_node);
311     unregister_kprobe(&kp_kmem_cache_alloc);
312     remove_proc_entry(PROC_FILENAME, NULL);
313
314     spin_lock(&mem_lock);
315     list_for_each_entry_safe(entry, tmp, &mem_alloc_list, list) {
316         list_del(&entry->list);
317         kfree(entry);
318     }

```

```

319     spin_unlock(&mem_lock);
320
321     pr_info("Memory_monitor_module_unloaded.\n");
322 }
323
324 module_init(mem_monitor_init);
325 module_exit(mem_monitor_exit);

```

Листинг 22 – Весь код загружаемого модуля ядра

```

1 #define pr_fmt(fmt) "ftrace_hook:_ " fmt
2
3 #include <linux/ftrace.h>
4 #include <linux/kallsyms.h>
5 #include <linux/kernel.h>
6 #include <linux/linkage.h>
7 #include <linux/module.h>
8 #include <linux/slab.h>
9 #include <linux/uaccess.h>
10 #include <linux/version.h>
11 #include <linux/kprobes.h>
12
13 MODULE_DESCRIPTION("Example_module_hooking_kmalloc()_and_
    kmem_cache_alloc()_via_ftrace");
14 MODULE_AUTHOR("Runov_Konstantin");
15 MODULE_LICENSE("GPL");
16
17 static unsigned long lookup_name(const char *name)
18 {
19     struct kprobe kp = {
20         .symbol_name = name
21     };
22     unsigned long retval;
23
24     if (register_kprobe(&kp) < 0) return 0;
25     retval = (unsigned long) kp.addr;
26     unregister_kprobe(&kp);
27     return retval;
28 }
29
30 /*
31  * There are two ways of preventing vicious recursive loops when

```

```

    hooking:
32 * - detect recursion using function return address
    (USE_FENTRY_OFFSET = 0)
33 * - avoid recursion by jumping over the ftrace call
    (USE_FENTRY_OFFSET = 1)
34 */
35 #define USE_FENTRY_OFFSET 0
36
37 /**
38 * struct ftrace_hook - describes a single hook to install
39 *
40 * @name:      name of the function to hook
41 *
42 * @function:  pointer to the function to execute instead
43 *
44 * @original:  pointer to the location where to save a pointer
45 *             to the original function
46 *
47 * @address:   kernel address of the function entry
48 *
49 * @ops:       ftrace_ops state for this function hook
50 *
51 * The user should fill in only &name, &hook, &orig fields.
52 * Other fields are considered implementation details.
53 */
54 struct ftrace_hook {
55     const char *name;
56     void *function;
57     void *original;
58
59     unsigned long address;
60     struct ftrace_ops ops;
61 };
62
63 static int fh_resolve_hook_address(struct ftrace_hook *hook)
64 {
65     hook->address = lookup_name(hook->name);
66
67     if (!hook->address) {
68         pr_debug("unresolved_symbol: %s\n", hook->name);
69         return -ENOENT;

```

```

70     }
71
72     *((unsigned long*) hook->original) = hook->address;
73
74     return 0;
75 }
76
77 static void notrace fh_ftrace_thunk(unsigned long ip, unsigned
    long parent_ip,
78     struct ftrace_ops *ops, struct ftrace_regs *fregs)
79 {
80     struct pt_regs *regs = ftrace_get_regs(fregs);
81     struct ftrace_hook *hook = container_of(ops, struct
        ftrace_hook, ops);
82
83     if (!within_module(parent_ip, THIS_MODULE))
84         regs->ip = (unsigned long)hook->function;
85 }
86
87 /**
88  * fh_install_hooks() - register and enable a single hook
89  * @hook: a hook to install
90  *
91  * Returns: zero on success, negative error code otherwise.
92  */
93 int fh_install_hook(struct ftrace_hook *hook)
94 {
95     int err;
96
97     err = fh_resolve_hook_address(hook);
98     if (err)
99         return err;
100
101     /*
102      * We're going to modify %rip register so we'll need IPMODIFY
        flag
103      * and SAVE_REGS as its prerequisite. ftrace's anti-recursion
        guard
104      * is useless if we change %rip so disable it with RECURSION.
105      * We'll perform our own checks for trace function reentry.
106      */

```

```

107     hook->ops.func = fh_ftrace_thunk;
108     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
109                     | FTRACE_OPS_FL_RECURSION
110                     | FTRACE_OPS_FL_IPMODIFY;
111
112     err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
113     if (err) {
114         pr_debug("ftrace_set_filter_ip()_failed:_%d\n", err);
115         return err;
116     }
117
118     err = register_ftrace_function(&hook->ops);
119     if (err) {
120         pr_debug("register_ftrace_function()_failed:_%d\n", err);
121         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
122         return err;
123     }
124
125     return 0;
126 }
127
128 /**
129  * fh_remove_hooks() - disable and unregister a single hook
130  * @hook: a hook to remove
131  */
132 void fh_remove_hook(struct ftrace_hook *hook)
133 {
134     int err;
135
136     err = unregister_ftrace_function(&hook->ops);
137     if (err) {
138         pr_debug("unregister_ftrace_function()_failed:_%d\n",
139                 err);
139     }
140
141     err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
142     if (err) {
143         pr_debug("ftrace_set_filter_ip()_failed:_%d\n", err);
144     }
145 }
146

```



```

147 /**
148  * fh_install_hooks() - register and enable multiple hooks
149  * @hooks: array of hooks to install
150  * @count: number of hooks to install
151  *
152  * If some hooks fail to install then all hooks will be removed.
153  *
154  * Returns: zero on success, negative error code otherwise.
155  */
156 int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
157 {
158     int err;
159     size_t i;
160
161     for (i = 0; i < count; i++) {
162         err = fh_install_hook(&hooks[i]);
163         if (err)
164             goto error;
165     }
166
167     return 0;
168
169 error:
170     while (i != 0) {
171         fh_remove_hook(&hooks[--i]);
172     }
173
174     return err;
175 }
176
177 /**
178  * fh_remove_hooks() - disable and unregister multiple hooks
179  * @hooks: array of hooks to remove
180  * @count: number of hooks to remove
181  */
182 void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
183 {
184     size_t i;
185
186     for (i = 0; i < count; i++)
187         fh_remove_hook(&hooks[i]);

```

```

188 }
189
190 /*
191  * Tail call optimization can interfere with recursion detection
192   based on
193  * return address on the stack. Disable it to avoid machine
194   hangups.
195  */
196 #if !USE_FENTRY_OFFSET
197 #pragma GCC optimize("-fno-optimize-sibling-calls")
198 #endif
199
200 static asmlinkage long (*real_sys_kmalloc)(struct pt_regs *regs);
201
202 static asmlinkage long fh_sys_kmalloc(struct pt_regs *regs)
203 {
204     long ret;
205
206     pr_info("kmalloc:_%zu\n", regs->di);
207     ret = real_sys_kmalloc(regs);
208
209     return ret;
210 }
211
212 static asmlinkage long (*real_sys_kmem_cache_alloc)(struct
213     pt_regs *regs);
214
215 static asmlinkage long fh_sys_kmem_cache_alloc(struct pt_regs
216     *regs)
217 {
218     long ret;
219
220     pr_info("kmem_cache_alloc:_%zu\n", regs->di);
221     ret = real_sys_kmem_cache_alloc(regs);
222
223     return ret;
224 }
225
226 #define SYSCALL_NAME(name) (name)
227
228 #define HOOK(_name, _function, _original) \

```

```

225     {
226         .name = SYSCALL_NAME(_name),
227         .function = (_function),
228         .original = (_original),
229     }
230
231 static struct ftrace_hook demo_hooks[] = {
232     HOOK("__kmalloc", fh_sys_kmalloc, &real_sys_kmalloc),
233     HOOK("kmem_cache_alloc", fh_sys_kmem_cache_alloc,
234         &real_sys_kmem_cache_alloc),
235 };
236
237 static int fh_init(void)
238 {
239     int err;
240
241     err = fh_install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
242     if (err)
243         return err;
244
245     pr_info("module_loaded\n");
246
247     return 0;
248 }
249
250 module_init(fh_init);
251
252 static void fh_exit(void)
253 {
254     fh_remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
255
256     pr_info("module_unloaded\n");
257 }
258
259 module_exit(fh_exit);

```

Листинг 23 – ftrace.c