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

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

Студент _____ ____К.А. Рунов____
(Подпись, дата) (И.О.Фамилия)

РЕФЕРАТ

Отчет 32 с., 2 рис., 0 табл., NNIGGERNIGGERNIGGERNIGGERNIGGERNIG
источн., NNIGGERNIGGERNIGGERNIGGERNIGGERNIGGERNIGGERNIGGERNIGGER
прил.

СОДЕРЖАНИЕ

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.

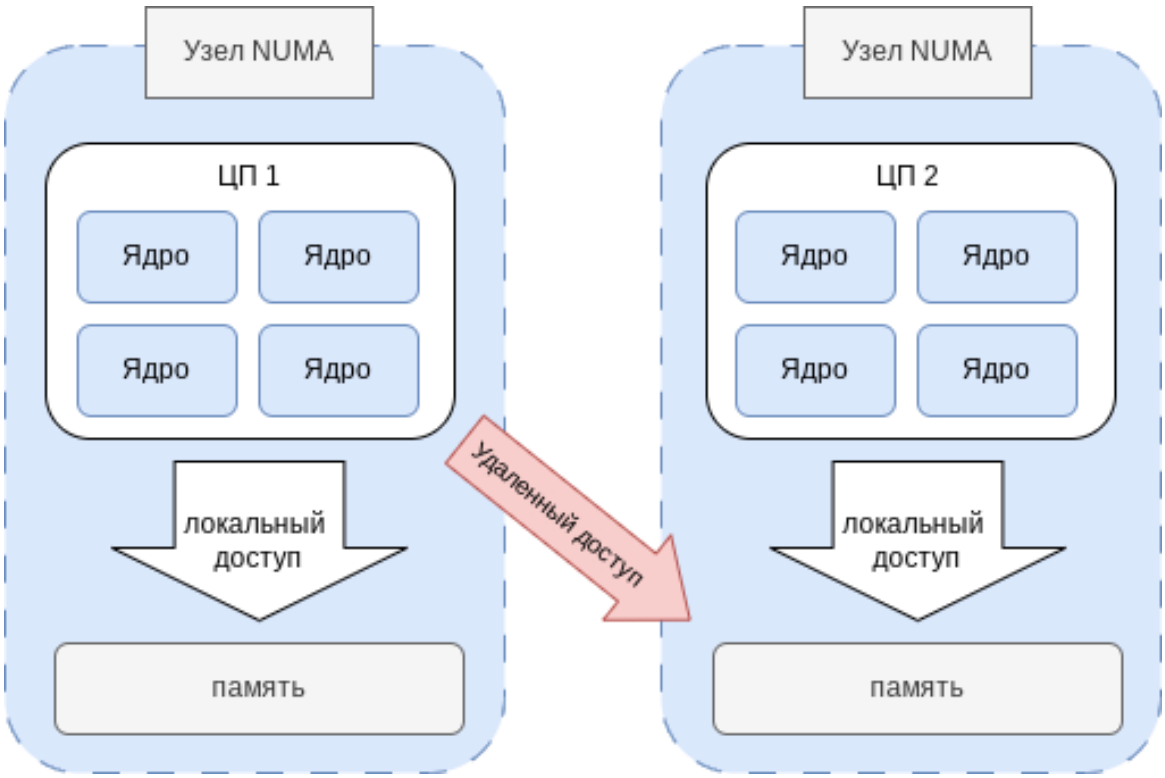

Рисунок 1 – Топология NUMA

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().
           That
7       * means the other users of this union MUST NOT use the bit to
8       * avoid collision and false-positive PageTail().
9       */
10     union {
11         struct {    /* Page cache and anonymous pages */
12             /**
13              * @lru: Pageout list, eg. active_list protected by
14              * lruvec->lru_lock.  Sometimes used as a generic list
15              * by the page owner.
16              */
17             union {
18                 struct list_head lru;
19                 /* Or, for the Unevictable "LRU list" slot */
20                 struct {
21                     /* Always even, to negate PageTail */
22                     void *__filler;
23                     /* Count page's or folio's mlocks */
24                     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                  unsigned long share;    /* share count for fsdax
                       */
35              };
36              /**
37               * @private: Mapping-private opaque data.
38               * Usually used for buffer_heads if PagePrivate.
39               * Used for swp_entry_t if PageSwapCache.
40               * Indicates order in the buddy system if PageBuddy.
41               */
42              unsigned long private;
43          };
44          ...
45          struct {    /* ZONE_DEVICE pages */
46              /** @pgmap: Points to the hosting device page map. */
47              struct dev_pagemap *pgmap;
48              void *zone_device_data;
49              /*
50               * ZONE_DEVICE private pages are counted as being
51               * mapped so the next 3 words hold the mapping, index,
52               * and private fields from the source anonymous or
53               * page cache page while the page is migrated to
                     device
54               * private memory.
55               * ZONE_DEVICE MEMORY_DEVICE_FS_DAX pages also
56               * use the mapping, index, and private fields when
57               * pmem backed DAX files are mapped.
58               */
59          };
60
61          /** @rcu_head: You can use this to free a page by RCU. */
62          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
                 number
68           * of times this page is referenced by a page table.
69           */
70          atomic_t _mapcount;
71          /*
72           * If the page is neither PageSlab nor mappable to
                 userspace,
73           * the value stored here may help determine what this page
74           * is used for.  See page-flags.h for a list of page types
75           * which are currently stored here.
76           */
77          unsigned int page_type;
78      };
79      ...
80      /*
81       * On machines where all RAM is mapped into kernel address
             space,
82       * we can simply calculate the virtual address. On machines
             with
83       * highmem some memory is mapped into kernel virtual memory
84       * dynamically, so we need a place to store that address.
85       * Note that this field could be 16 bits on x86 ... ;)
86       *
87       * Architectures with slow multiplication can define
88       * WANT_PAGE_VIRTUAL in asm/page.h
89       */
90  #if defined(WANT_PAGE_VIRTUAL)
91      void *virtual;          /* Kernel virtual address (NULL if
92                                 not kmapped, ie. highmem) */
93  #endif /* WANT_PAGE_VIRTUAL */
94      ...
95  };
```

Листинг 2 – struct page

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

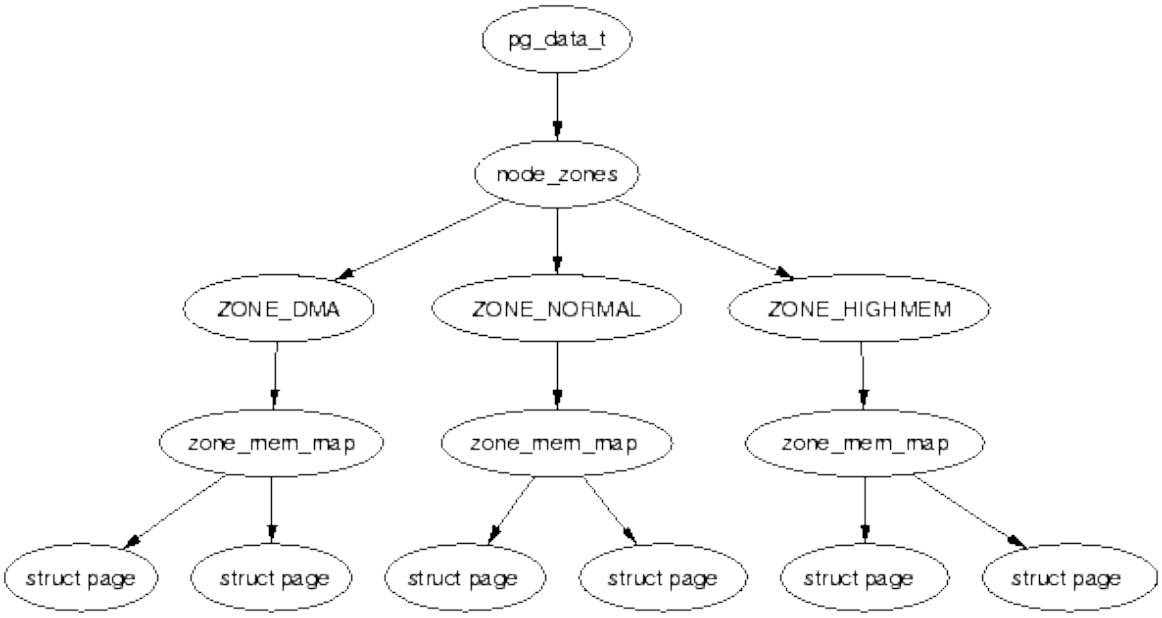

Рисунок 2 – Отношение между узлами памяти, зонами и страницами [2]

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;

— Создаёт struct kmem_cache, которая управляет списком объектов;

— Создаёт 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 (size $\leq$ PAGE_SIZE) выделение происходит через SLAB, а для больших объектов (size $>$ PAGE_SIZE) выделение происходит через alloc_pages().

Ниже представлены листинги функций kmalloc, __kmalloc и kmalloc_large.

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

Листинг 5 – kmalloc

```
void *__kmalloc(size_t size, gfp_t flags)
{
    struct kmem_cache *cache;
    cache = kmalloc_slab(size, flags);
    if (unlikely(!cache))
        return kmalloc_large(size, flags);
    return kmem_cache_alloc(cache, flags);
}
```

Листинг 6 – __kmalloc

```
static void *kmalloc_large(size_t size, gfp_t flags)
{
    struct page *page;
    page = alloc_pages(flags, get_order(size));
    if (!page)
        return NULL;
    return page_address(page);
}
```

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

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

Листинг 8 – Выделение памяти в SLAB-кэше

```
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 выполниться 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
           disarmed */
8      unsigned long nmissed;
9
10     /* location of the probe point */
11     kprobe_opcode_t *addr;
12
13     /* Allow user to indicate symbol name of the probe point */
14     const char *symbol_name;
15
16     /* Offset into the symbol */
17     unsigned int offset;
18
19     /* Called before addr is executed. */
20     kprobe_pre_handler_t pre_handler;
21
22     /* Called after addr is executed, unless... */
23     kprobe_post_handler_t post_handler;
24
25     /* Saved opcode (which has been replaced with breakpoint) */
26     kprobe_opcode_t opcode;
27
28     /* copy of the original instruction */
29     struct arch_specific_insn ainsn;
30
31     /* Indicates various status flags.
32      * Protected by kprobe_mutex after this kprobe is registered.
33      */
34     u32 flags;
35 };
```

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: size = %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 tracepoint «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 tracepoint. 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

perf и eBPF

1.4.2 Системные интерфейсы ядра для мониторинга памяти

/proc/meminfo

/proc/slabinfo

1.4.3 Сравнительный анализ подходов

1.4.4 Выбор метода мониторинга

Вывод

2 Конструкторский раздел

3 Технологический раздел

4 Исследовательский раздел

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).

ПРИЛОЖЕНИЕ А