Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии**»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К НАЧУНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

Классификация методов распределения памяти в ядре Linux

Студент группы **ИУ7-12М** _____ **А. В. Романов**

(Подпись, дата) (И.О. Фамилия)

Руководитель НИР _____ **А. А. Оленев**

(Подпись, дата) (И.О. Фамилия)

***2022** г.*

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Динамическая память – ценный вычислительный ресурс, необходимый для управления всей системой. Производительность системы зависит от эффективности управления динамической памятью. Поэтому, все современные многозадачные операционные системы пытаются оптимизировать использование оперативной памяти выделяя ее только в случае необходимости и освобождая ее при первой же возможности. Для достижения данной цели, ядро операционной системы использует специальные алгоритмы и структуры данных. Цель данной работы – классификация методов распределения памяти в ядре Linux.

Для достижения поставленной цели необходимо решить следующие задачи:

- провести обзор существующих методов распределения памяти в ядре Linux;
- описать плюсы и недостатки каждого из методов;
- сформулировать критерии классификации;
- классифицировать существующие методы распределения памяти;

1 Анализ предметной области

В этом разделе будут проведен анализ предметной области: даны понятия ядра Linux, страниц памяти и механизма виртуальной памяти.

1.1 Ядро Linux

Ядро Linux [1] – ядро операционной системы с открытым исходным кодом, распространяющееся как свободное программное обеспечение. Ядро Linux является монолитным, то есть содержит в себе все функции, возлагаемые на операционную систему и работает в одном адресном пространстве. При разработке ядра из микроядерной модели были позаимствованы некоторые решения: модульный принцип построения, приоритетное планирование самого себя, поддержка многопоточного режима и динамической загрузки в ядро внешних бинарных файлов (модулей ядра) [2].

Ниже представлены отличительные черты ядра Linux:

- потоки ничем не отличаются от обычных процессов. С точки зрения ядра все процессы одинаковы и лишь некоторые из них делят ресурсы между собой;
- динамическая загрузка модулей ядра;
- приоритетное планирование. Ядро способно прервать выполнение текущего процесса, даже если оно выполняется в режиме ядра;
- объектно-ориентированная модель устройств. Ядро поддерживает классы устройств и события;
- ядро Linux является результатом свободной и открытой модели разработки [2].

1.2 Страницы памяти

Работа с памятью в ядре Linux организована с помощью примитива страниц. Страница памяти – набор из n байт, непрерывно расположенных в памяти. Наименьшими адресуемыми единицами памяти в системе являются байт и машинное слово, но такой подход не является удобным [2]. Размер страницы

это аппаратно зависимая, фиксированная константа `PAGE_SIZE`, объявленная в исходном коде ядра Linux. На большинстве современных архитектур, поддерживаемых Linux, размер страницы составляет 4 Кб [3]. Страничная организация памяти позволяет реализовать механизм виртуальной памяти.

В ядре Linux каждая физическая страница памяти описывается с помощью структуры `struct page`, которая имеет большое количество полей, часть из которых используется по разному, в зависимости от текущей ситуации.

1.3 Виртуальная память

Виртуальная память – специальный механизм организации памяти, при котором процессы работают не с физическими адресами напрямую, а с линейными. Трансляция линейных адресов в физические чаще всего реализована аппаратно, с помощью блока управления памяти (англ. memory managment unit – MMU [4]). Кроме того, существует аппаратный способ ускорения трансляции – буффер ассоциативной трансляции (англ. translation lookaside buffer – TLB [5]). В связи с этим, можно сделать вывод о относительной быстроте трансляции адресов, что никак не сказывается на отзывчивости системы.

Механизм виртуальной памяти обладает следующими преимуществами:

- изоляция пользовательских процессов друг от друга;
- возможность использовать больше оперативной памяти, чем её физически установлено в системе;
- нет необходимости управлять общим адресным пространством.

Ядро Linux поддерживает механизм виртуальной памяти и реализовывает его с помощью таблиц страниц. Данная структура данных позволяет из линейного адреса ячейки памяти получить физический. Схема работы таблиц страниц представлена на рисунке 1.

Для преобразования линейного адреса в физический, ядро Linux использует 3 каталога и таблицу страниц:

- глобальный каталог страниц;
- верхний каталог страниц;

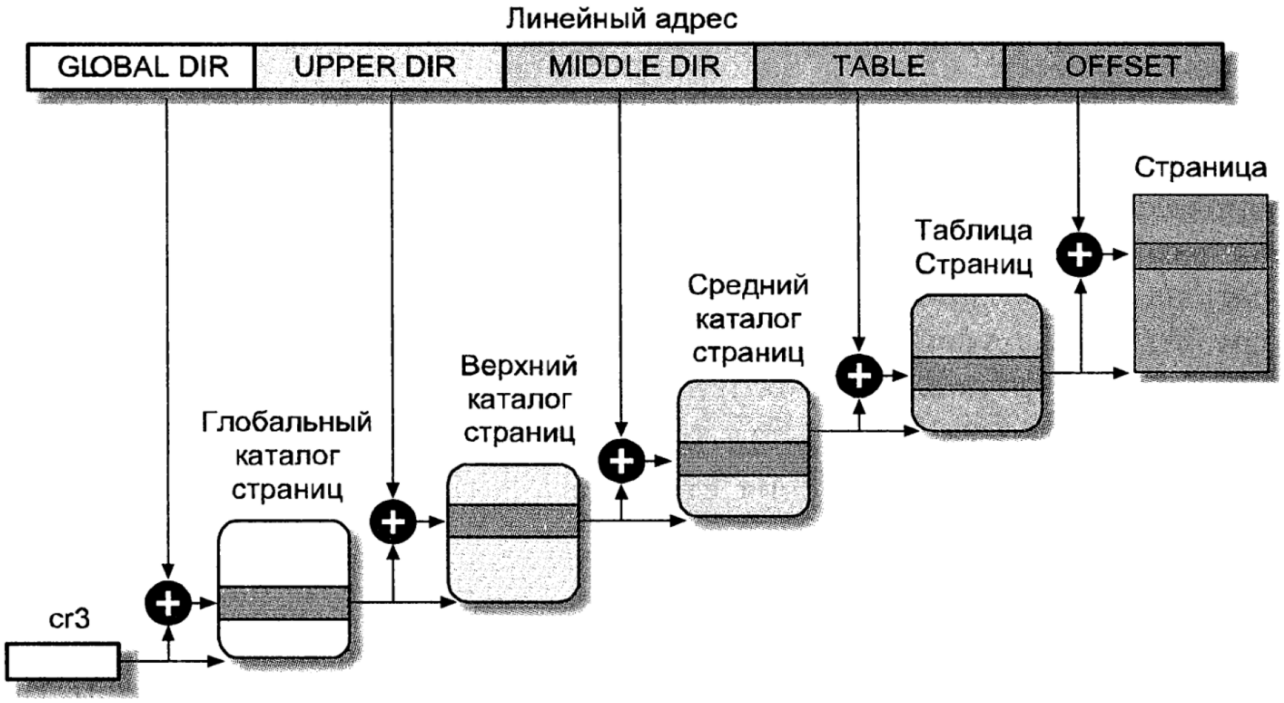

Рисунок 1 – Трансляция линейного адреса в физический

- средний каталог страниц;
- таблица страниц;

Элементами этих каталогов являются адреса, указывающие на начало соответствующего каталога. Глобальный каталог указывает на верхний каталог, верхний на средний, средний на таблицу страниц, а таблица страница на физический адрес искомой страницы. В самом линейном адресе находятся смещения от начала соответствующего каталога. Адрес глобального каталога страниц обычно хранится в некотором регистре – например, на архитектуре x86, этот регистр называется CR3. Каждый процесс в системе имеет свою собственную таблицу страниц.

2 Существующие методы распределения памяти

В данном разделе будут рассмотрены методы распределения памяти, представленные в ядре Linux. Рассмотренные методы распределения памяти будут классифицированы, сделаны вывод о подходящей сфере задач, в которых рассмотренные методы будут наиболее эффективны.

2.1 Алгоритм «Buddy-система»

При частом выделении групп смежных страниц физической памяти ядро должно придерживаться устойчивой и эффективной стратегии. Наиболее известная проблема, с которой сталкивается ядро решая данную задачу, называется внешней фрагментацией. Суть заключается в том, что частые запросы на выделение и освобождение смежных групп страниц разного размера приводит к ситуации, когда несколько небольших участков памяти свободных страниц находятся внутри выделенных страниц. Это может привести к тому, что в системе есть запрашиваемое количество свободных страниц, но она всё равно не может удовлетворить запрос.

Существует два метода борьбы с внешней фрагментацией:

- аппаратный – использование электронной схемы, позволяющей отображать группы несмежных свободных страниц в непрерывные интервалы линейных адресов;
- программный – использовать алгоритм, позволяющий избегать разбиения большого свободного блока страниц ради удовлетворения запроса на меньший блок.

Ядро Linux решает проблему внешней фрагментации с помощью алгоритма buddy-системы. Все свободные страниц памяти объединяются в 11 списков, содержащие блоки состоящие из 2^k смежных страниц, где $k = 0..10$. Физический адрес самого первого страничного кадра в блоке кратен размеру группы: $address \bmod k = 0$. Таким образом, количество запрашиваемых страниц у аллокатора, использующего алгоритм buddy-систем, должно быть кратно степеням двойки.

При запросе группы, состоящей из 2^k смежных страниц памяти, алгоритм проверяет наличие свободного блока в списке блоков, которые содержат 2^k смежных страниц памяти. Если блок найден, алгоритм возвращает адрес первой смежной страницы памяти из найденного блока, удаляет блок из свободного

списка и завершает свою работу. В обратном случае (если такой блок не найден), алгоритм продолжает поиск свободного блока в списке из $2^(k+1)$ смежных страниц памяти. В случае успеха, алгоритм переносит блок из 2^k страниц в соответствующий ему список, а оставшиеся 2^k страниц выделяет для удовлетворения запроса. Если блок не найден, алгоритм продолжает поиск в списке из $2^(k+2)$ смежных страниц, в случае успеха итеративно перенося часть страниц в соответствующие списки блоков. Если не в одном из списков не было найдено ни одного свободного блока, алгоритм завершается и сигнализирует об ошибке.

При освобождении блоков страниц, ядро пытается слить пары buddy-блоков размера S в один блок размера $2S$ (отсюда и взялось название buddy (приятель)). Два блока считаются buddy-блоками, если:

- оба блока имеют одинаковый размер S;
- они являются смежными физическими;
- выполняется условие $address\ mod\ (2 * S) = 0$, где $address$ – физический адрес первой страницы в первом блоке.

Ниже приведем пример работы алгоритма при запросе $2^0 = 1$ страниц (см. рисунки 2 - 5)

1. Алгоритм обошёл все списки свободных блоков начиная с $k = 0$, найдя свободный блок в списке с $k = 3$. В данном списке оказался один свободный блок, отмеченный голубым цветом (см. рис 2).

2. Далее, алгоритм разбивает блок из списка с $k = 3$ на два блока с $k = 2$, добавляя их в соответствующий список. Блоки с $k = 2$ на рисунке 3 отмечены жёлтым цветом.

3. Разбиение продолжается: один из блоков из списка с $k = 2$ разбивается на два блока с $k = 1$, которые отмечены на рисунке 2 фиолетовым цветом.

4. Наконец, разбив один из блоков из списка с $k = 1$ на два блока с $k = 0$, помеченные красным цветом на рисунке 5, алгоритм находит подходящую смежную область, отмеченную зеленым цветом, (в данном примере это одна страница, т.к. $k = 0$) и завершает свою работу.

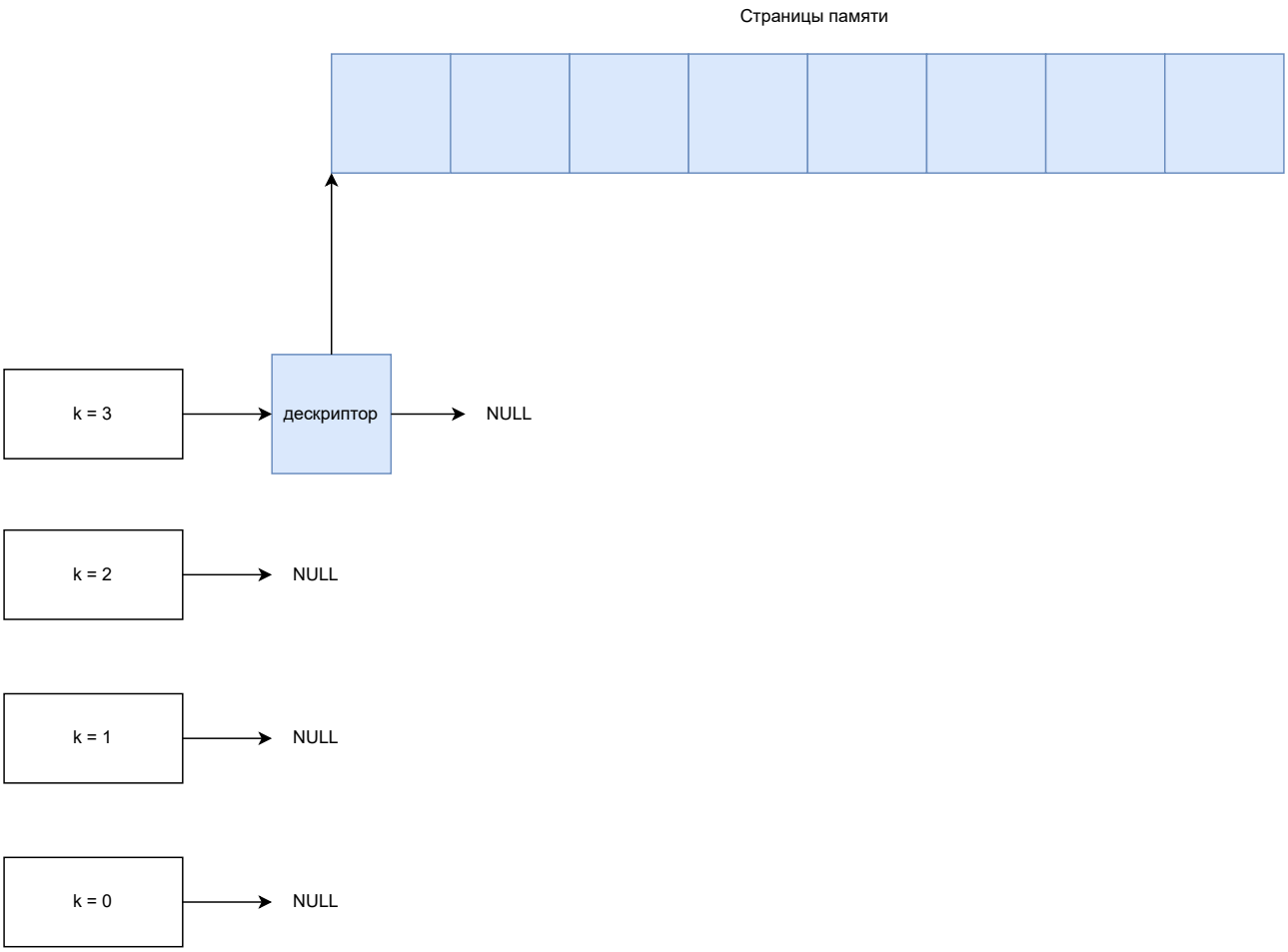

Рисунок 2 – Пример работы алгоритма buddy-системы, часть 1

При попытке освобождения блока работа алгоритма аналогична, но выполняется наоборот. В конечном итоге, при освобождении ранее запрошенного у системы зеленого блока (см. рисунок 5), в системе останется один свободный блок с $k = 3$ (см. рисунок 2).

2.1.1 Реализация в ядре Linux

Реализация в ядре Linux построена на двух структурах данных:

- `mem_map` – глобальная переменная, массив типа `struct page`, описывающая всю физическую память в системе. Первый элемент массива описывает первую физической странице памяти, второй - вторую, и так далее.
- массив структур типа `struct free_area`.

Массив, состоящий из 11 элементов и указывающий на списки блоков (дескрипторы свободных смежных областей памяти), описывается с помощью

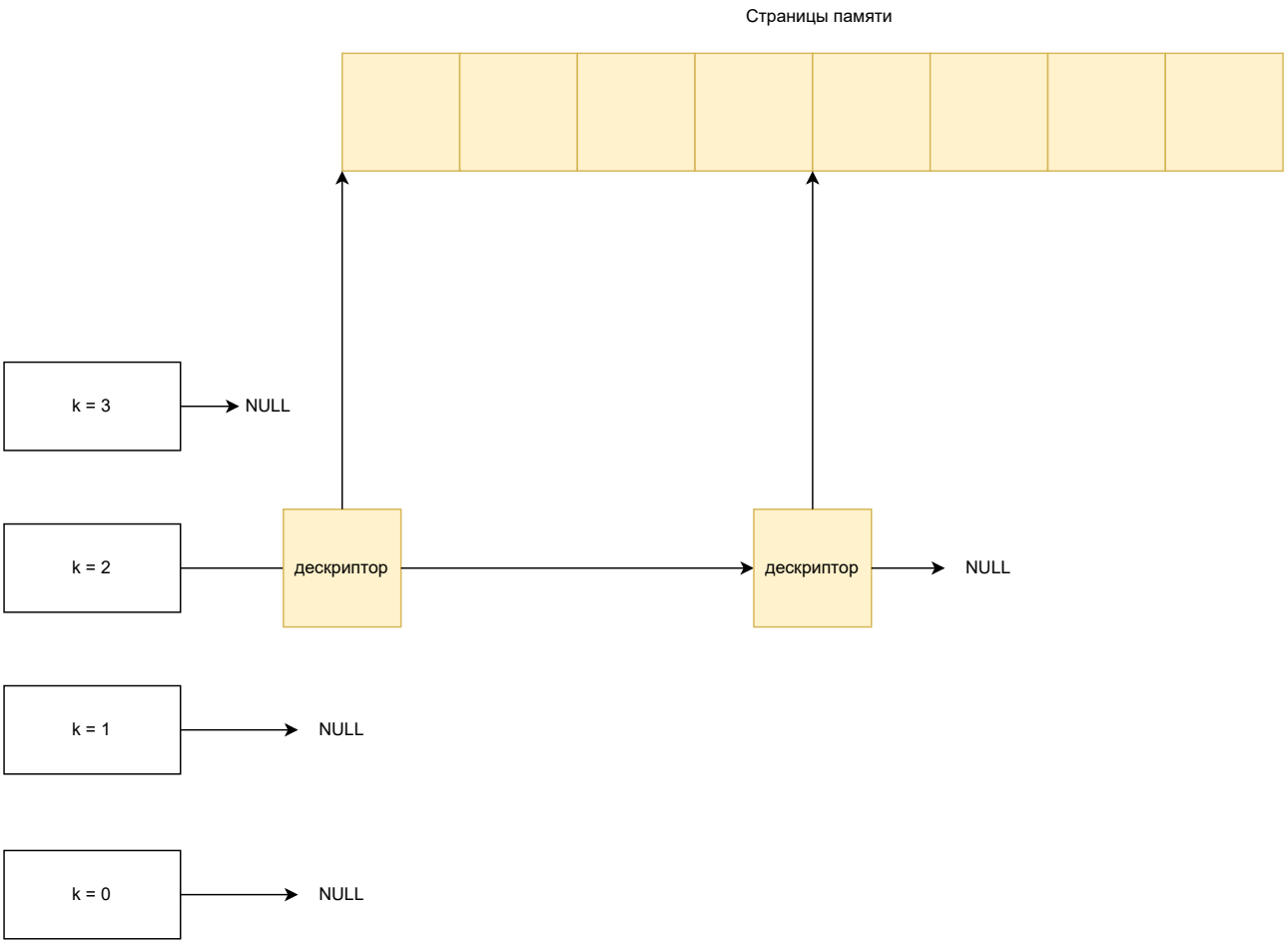

Рисунок 3 – Пример работы алгоритма buddy-системы, часть 2

массива структур типа `struct free_area`. Объявление данной структуры представлено в листинге 1.

Листинг 1: Структура `struct free_area`

```
1  struct free_area {
2      struct list_head    free_list[MIGRATE_TYPES];
3      unsigned long        nr_free;
4  };
```

Структура содержит внутри себя два поля:

- `free_list` – список всех блоков (дескрипторов), в том числе занятых, указывающие на соответствующие им смежные участки памяти из 2^k сво-

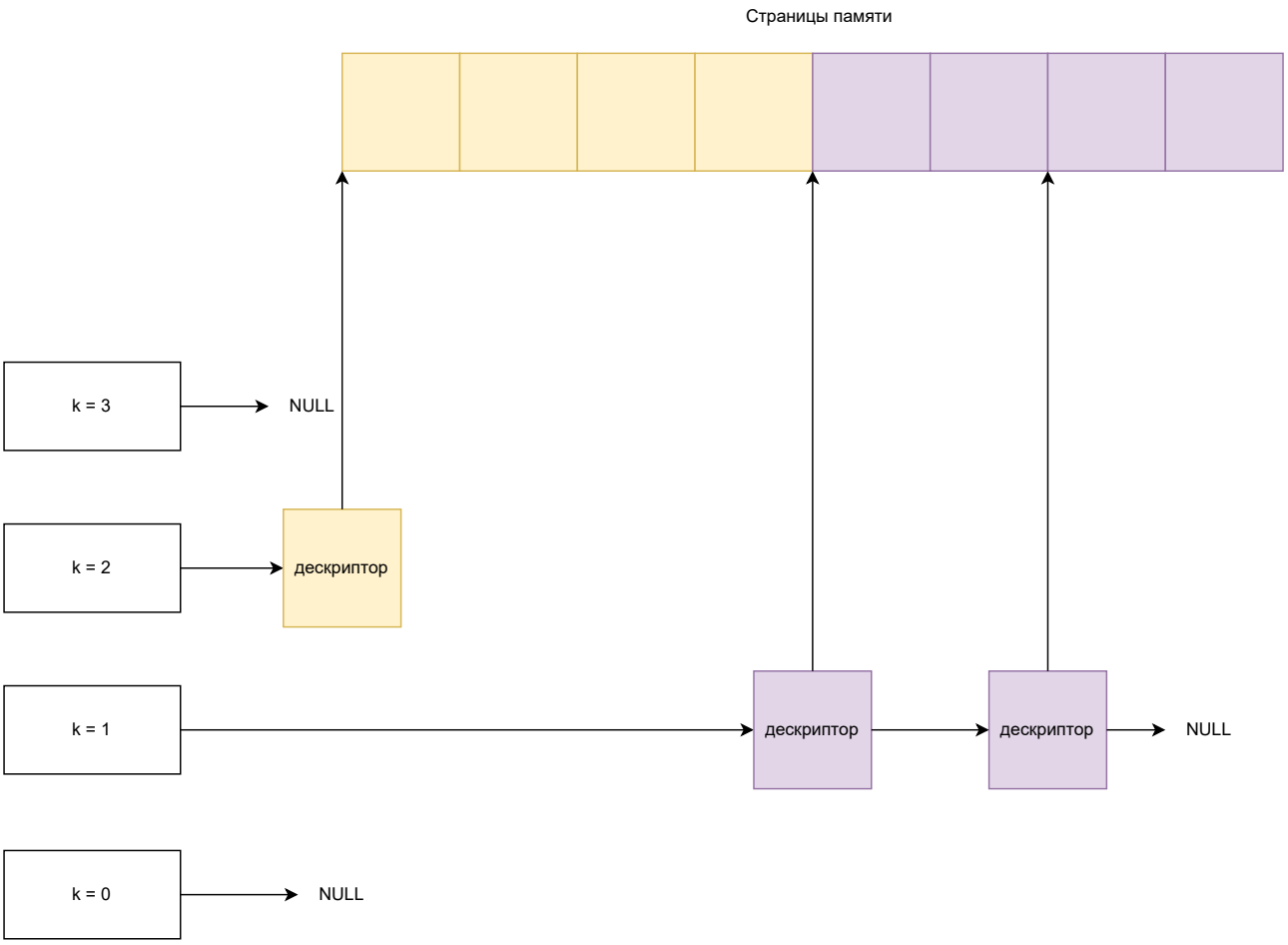

Рисунок 4 – Пример работы алгоритма buddy-системы, часть 3

бодных страниц;

- `nr_free` – количество свободных блоков в списке.

Выделение блоков возможно с помощью функции `alloc_pages`, а освобождение используя `free_pages`. Сигнатуры этих функций представлены в листингах 2 - 3.

Листинг 2: Сигнатура функции `alloc_pages`

```
1  struct page *alloc_pages(gfp_t gfp, unsigned int order);
```

Функция `alloc_pages` принимает два параметра:

- `gfp` – набор флагов, используемых при аллокации. Например, при указании флага `GFP_ATOMIC`, что запрос на выделения памяти должен пройти

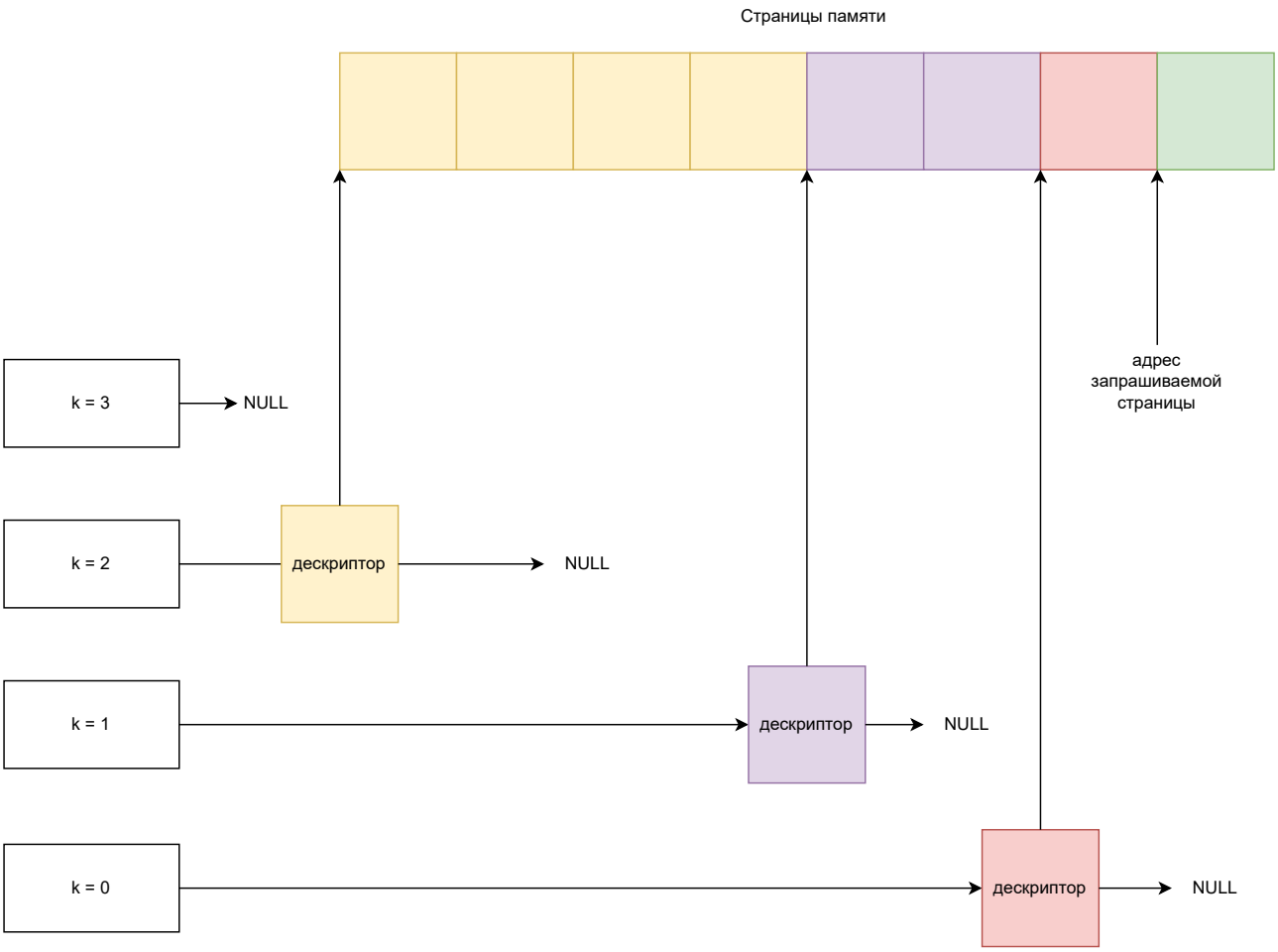

Рисунок 5 – Пример работы алгоритма buddy-системы, часть 4

атомарно, что означает что вызывающий тракт ядра не может быть блокирован, либо завершиться неудачей.

- `order` – порядок запрашиваемого смежного блока страниц, т.е. $S = 2^k$, где S – количество физических смежных страниц памяти.

Функция возвращает указатель на дескриптор типа `struct page` первой страницы из запрашиваемого участка смежных страниц.

Листинг 3: Сигнатура функции `free_pages`

```
1  void free_pages(unsigned long addr, unsigned int order);
```

Функция `free_pages` так же принимает два параметра:

- `addr` – физический адрес первой страницы освобождаемого блока смежных страниц;
- `order` – порядок освобождаемого блока.

Реализация алгоритма выделения и освобождения смежных блоков страниц памяти в ядре Linux аналогична алгоритму описанному в прошлой главе. Алгоритм работает с физически смежными страница памяти, например, при $order = 3$, алгоритм выделит блок из $2^3 = 8$ физически смежных страниц оперативной памяти. Рассматриваемый метод работает со страницами памяти, то есть минимум он сможет выделить `PAGE_SIZE` (обычно 4096) байт. При необходимости использовать менее чем `PAGE_SIZE` байт, часть памяти будет не использована вовсе. Использовать страницу целиком, для хранения нескольких байтов – расточительно.

2.2 Управление областями памяти

Область памяти – последовательность ячеек, имеющих смежные физические адреса и произвольную длину. Buddy-аллокации не подходит для выделения таких областей памяти, потому что работает со страницами, размер которых статичен.

В ядре Linux принят следующий подход к управлению такими областями: размер областей привязан к степеням двойки, размеры расположены в диапазоне от 32 до 131 072 (геометрическая прогрессия). Такой подход позволяет бороться с внутренней фрагментацией (несоответствие между размером запрашиваемой памяти к выделенной для удовлетворения запроса) и гарантирует что она всегда будет меньше 50%. То есть, при запросе k байт, число k будет округлено до ближайшей (в большую сторону) степени двойки.

2.2.1 Slab аллокатор

Функции ядра многократно запрашивают области памяти одного и того же размера, например, под дескрипторы (структуры) описывающие что-либо. Когда ядро создаёт новый процесс, оно выделяет области памяти под несколько структур фиксированного размера: дескриптор процесса, дескриптор файла и

так далее.

Запросы на участки памяти фиксированного размера, про которые известно, что они возникают часто, можно эффективно обрабатывать создав пулл из объектов имеющих соответствующий размер. Такой способ помогает избежать внутренней фрагментации. Редкие запросы на участки памяти, можно обрабатывать с использованием схемы, основанной на применении объектов с геометрически распределенными размерами, как было описано ранее.

Slab аллокатор группирует объекты в кэши, каждый кэш предназначен для выделение объектов одного типа, и, соответственно, одного и того же размера. Например, для выделения участка памяти, в котором будет располагаться структура `struct file`, будет использован один кэш, а для структуры `struct task_struct` будет использован другой кэш.

В ядре Linux каждый slab кэш описывается с помощью структуры `struct kmem_cache`, описание которой, с наиболее важными полями, представлено в листинге 4.

Листинг 4: Структура `struct kmem_cache`

```
1  struct kmem_cache {
2      unsigned int size;
3      slab_flags_t flags;
4      unsigned int num;
5
6      const char *name;
7      struct list_head list;
8      int refcount;
9      int object_size;
10     int align;
11
12     struct kmem_cache_node *node[MAX_NUMNODES];
13 };
```

Подробное описание полей структуры `struct kmem_cache`:

- `size` – общий размер кэша;
- `flags` – флаги, используемые при аллокации;
- `num` – количество объектов, хранящихся в кэше;
- `name` – строка, имя кэша;
- `list` – циклический двунаправленный список всех кэшей в системе;
- `refcount` – количество ссылок на кэш в системе. В случае, если `refcount` = 0, система удалит кэш.
- `object_size` – размер объектов, хранящихся в кэше;
- `align` – выравнивание объектов в памяти;
- `node` – массив структур типа `struct kmem_cache_node` (см. листинг 5)

Таким образом, структура `struct kmem_cache` описывает кэш в целом, и хранит указатель на структуру `struct kmem_cache_node`, описанную в листинге 5.

Листинг 5: Структура `struct kmem_cache_node`

```
1  struct kmem_cache_node {
2      spinlock_t list_lock;
3
4      struct list_head slabs_partial;
5      struct list_head slabs_full;
6      struct list_head slabs_free;
7      unsigned long total_slabs;
8      unsigned long free_slabs;
9      unsigned long free_objects;
10 };
```

Структура `struct kmem_cache_node` хранит указатели на списки трех типов:

- `slabs_partial` – в этом списке хранятся дескрипторы участков памяти как со свободными, так и с несвободными объектами;
- `slabs_full` – дескрипторы полностью заполненных участков памяти;
- `slabs_free` – дескрипторы полностью свободных участков памяти.

В рассматриваемой структуре так же хранятся дополнительные поля, упрощающие работу с вышеописанными списками:

- `total_slabs` – количество все дескрипторов;
- `free_slabs` – количество свободных дескрипторов;
- `free_objects` – общее количество всех свободных объектов.

Каждый участок памяти в кэше имеет собственный дескриптор типа `struct slab`, описание которого приводится в листинге 6.

Листинг 6: Структура `struct slab`

```
struct slab {
    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
    };

    struct kmem_cache *slab_cache;
    void *freelist;
    void *s_mem;

    struct {
        unsigned inuse:16;
        unsigned objects:15;
        unsigned frozen:1;
    };
};
```

Данная структура содержит в себе несколько полей:

- `slab_list` – указатели, используемые в одном из трех двунаправленных списокв дескрипторов участков памяти (см. листинг **??**);
- `slab_cache` – указатель на структуру типа `struct kmem_cache`, описанную в листинге 4;
- `freelist` – массив индексов всех свободных объектов;
- `s_mem` – адрес первого объекта в участке;
- `inuse` – количество несвободных объектов в участке памяти;

Взаимосвязи между структурами, которые были описаны выше, представлены на рисунке 6. Красным цветом отмечены занятые объекты, зеленным – свободные.

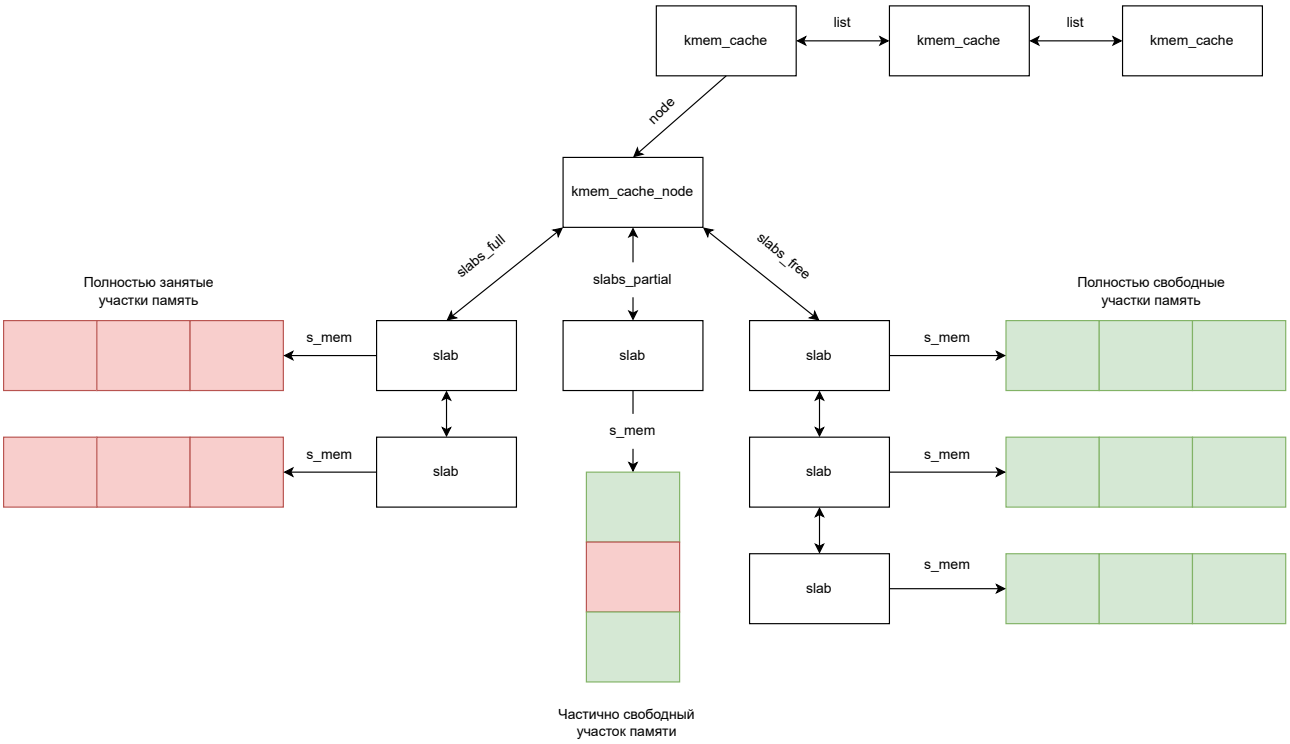

Рисунок 6 – Взаимосвязи между структурами данных, используемые в работе Slab аллокатора.

На рисунке 7 показана взаимосвязь между дескрипторами участков (`struct slab`) и объектами в памяти. Красным цветом отмечены занятые объекты, зеленным – свободные.

Для работы со slab аллокатором в ядре Linux используется API, представленное в листинге 7

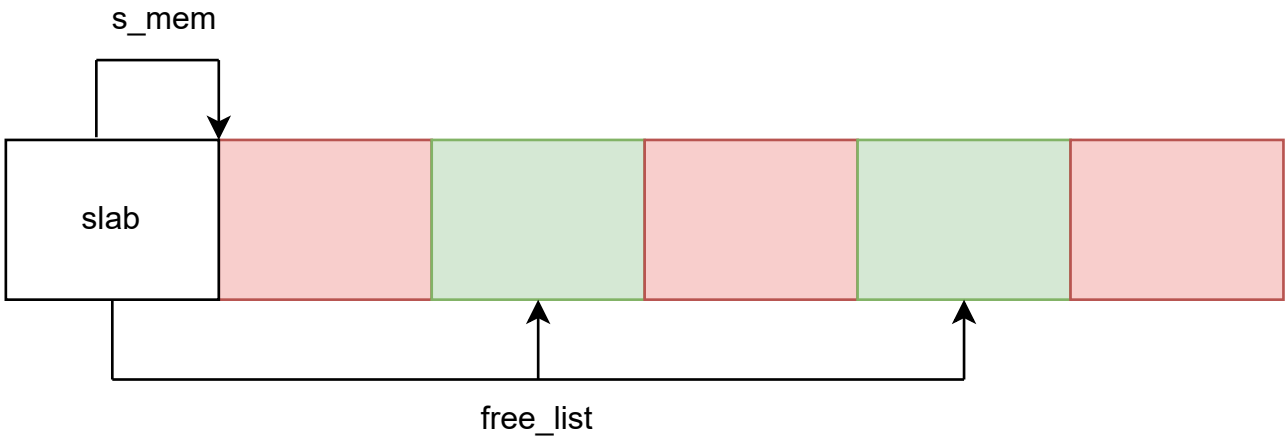

Рисунок 7 – Описание работы структуры `struct slab`.

Листинг 7: API для работы с Slab кэшем в ядре Linux

```
1  struct kmem_cache *
2  kmem_cache_create(const char *name, unsigned int size,
3    unsigned int align, slab_flags_t flags, void (*ctor)(void *))
4
5  void *kmem_cache_alloc(struct kmem_cache *s, gfp_t flags);
6
7  void kmem_cache_free(struct kmem_cache *s, void *objp);
8
9  void kmem_cache_destroy(struct kmem_cache *s);
```

- `kmem_cache_create` – создать Slab кэш с именем `name` размером `size`. Возвращает указатель на структуру типа `struct kmem_cache`.
- `kmem_cache_alloc` – запросить объект в Slab кэше `s`;
- `kmem_cache_free` – освободить объект `objp` в кэше `s`;
- `kmem_cache_destroy` – уничтожить кэш `s`;

Функции, описанные выше, используются для создания кэша, который хранит объекты определенного размера. В случае, если системе требуются участки памяти разной длинны, и такие запросы редки, следует использовать пару

функции kmalloc и kfree, сигнатуры которых представлены в листинге 8.

Листинг 8: Сигнатуры функций kmalloc и kfree

```
1  void *kmalloc(size_t size, gfp_t gfp);
2
3  void kfree(const void *object)
```

Функция kmalloc так же использует Slab аллокатор, обрабатывая запрос с помощью группы общих кэшей размером от 32 до 131 072 байт. Таким образом, размер области памяти, запрашиваемый у системы округляется к ближайшей степени двойки.

2.2.2 Пулы памяти

Пул памяти – резерв динамической области памяти, выделенной для конкретного компонента ядра, то есть использовать память может только тот компонент, который этот пул создал. Обычно, такой пул памяти создается в помощь slab аллокатору, т.е. используется в качестве резерва объектов для участков памяти. В ядре Linux такой пул описывается с помощью структуры struct mempool_t, поля которой представлены в листинге 9.

- min_nr – максимальное количество элементов в пуле;
- curr_nr – текущее количество элементов в пуле;
- elements – указатель на массив указателей на зарезервированные элементы;
- pool_data – специальные данные, используемые владельцем пула;
- alloc – метод для выделения памяти;
- free – метод для освобождения элемента;
- wait – очередь, используемая, когда пул памяти пуст.

Листинг 9: Структура `mempool_t`

```
typedef struct mempool_s {
    spinlock_t lock;
    int min_nr;
    int curr_nr;
    void **elements;

    void *pool_data;
    mempool_alloc_t *alloc;
    mempool_free_t *free;
    wait_queue_head_t wait;
} mempool_t;
```

Суть работы с пулом памяти заключается в том, что все запросы на выделение объектов в памяти происходит с помощью функции `mempool_alloc`, которая в свою очередь сначала вызывает метод `alloc`, и, только в случае неудачи, берет объект необходимого размера из зарезервированного пула памяти. Обычно, в качестве методов `alloc` и `free` используются функции `kmem_cache_alloc` и `kmem_cache_free`, которые были описаны ранее.

Использование пулов памяти может быть выгодным для компонентов ядра, которые должны быть отказоустойчивыми в ситуациях нехватки памяти. В случаях, когда динамической памяти становится настолько мало, что все выделения обречены на провал, а компонент должен продолжать свою работу без ошибок, необходимо использовать пулы памяти.

2.2.3 zsmalloc

zsmalloc – специально спроектированный аллокатор для работы в ситуациях, когда в системе критически мало памяти. Данный аллокатор так же работает с физически непрерывными участками памяти, но максимальный участок

памяти, который он может выделить, ограничен размером PAGE_SIZE. Данный аллокатор использует функцию alloc_page, то есть как и Slab кэши, базируется поверх алгоритма buddy-системы. Аллокатор предоставляет специальное API, описанное в листинге 10

Листинг 10: API для работы с zsmalloc

```
1  struct zs_pool *zs_create_pool(const char *name);
2
3  void zs_destroy_pool(struct zs_pool *pool);
4
5  unsigned long zs_malloc(struct zs_pool *pool, size_t size,
6    gfp_t flags);
7
8  void zs_free(struct zs_pool *pool, unsigned long obj);
9
10 void *zs_map_object(struct zs_pool *pool,
11   unsigned long handle, enum zs_mapmode mm);
12
13 void zs_unmap_object(struct zs_pool *pool,
14   unsigned long handle);
```

Рассмотрим подробнее данные функции:

- zs_create_pool – создать пулл, в котором в дальнейшем будут выделяться объекты;
- zs_destroy_pool – уничтожить пулл объектов;
- zs_malloc – выделить объект размером size внутри пулла pool. Возвращает целое без знаковое число, обычно именуемое handle;
- zs_free – освободить ранее выделенный функцией zs_malloc объект;
- zs_map_object – получить соответствие между числом (handle), ко-

торое вернула функция `zs_malloc` и указателем на выделенную областью памяти, то есть получить указатель на начало выделенного аллокатором участка памяти. Из-за внутренних особенностей архитектуры zsmalloc, в один момент времени может быть получено не более одного соответствия между `handle` и указателем на выделенную область памяти;

- `zs_unmap_object` – убрать соответствие между `handle` и адресом на выделенную акллокатором память. После вызова этой функции, обращаться к ранее выделенному участку памяти запрещено.

Единицей диспетчеризации аллокатора является структура данных, называемая zspage, которая описывается соответствующей структурой `struct zspage`, поля которой представлены в листинге 11.

Листинг 11: Структура `struct zspage`

```
1  struct zspage {
2      struct {
3          unsigned int huge:HUGE_BITS;
4          unsigned int fullness:FULLNESS_BITS;
5          unsigned int class:CLASS_BITS + 1;
6          unsigned int isolated:ISOLATED_BITS;
7          unsigned int magic:MAGIC_VAL_BITS;
8      };
9      unsigned int inuse;
10     unsigned int freeobj;
11     struct page *first_page;
12     struct list_head list;
13     struct zs_pool *pool;
14 };
```

zspage – это связанный список из объеденных страниц `struct page`, которые указывают друг на друга с помощью поля `index`. zspage – это струк-

тура данных, которая предоставляет хранилище для объектов одинакового размера, размещенных в памяти друг за другом, основываясь на структуре `struct page`. Таким образом, объект может находиться сразу на двух страница одновременно: часть байт на странице с индексом i, а оставшаяся часть на странице $i + 1$. Суть заключается в том, чтобы подобрать такое количество страниц памяти `struct page`, чтобы максимально вместительно разместить в памяти объекты размером n.

Рассмотрим подробнее, поля структуры описанные в листинге 11:

- с помощью анонимный структуры (битовое поле), описываются различные флаги необходимые для работы;
- `inuse` – количество занятых объектов на странице `zspage`;
- `freeobj` – количество свободных объектов;
- `first_page` – указатель на первую странице в связанном списке страниц;
- `list` – список структур `zspage`;
- `pool` – указатель на пул страниц, описание которого приводится далее.

На рисунке 8 представлена схема взаимодействия структуры `struct zspage` и структур `struct page`.

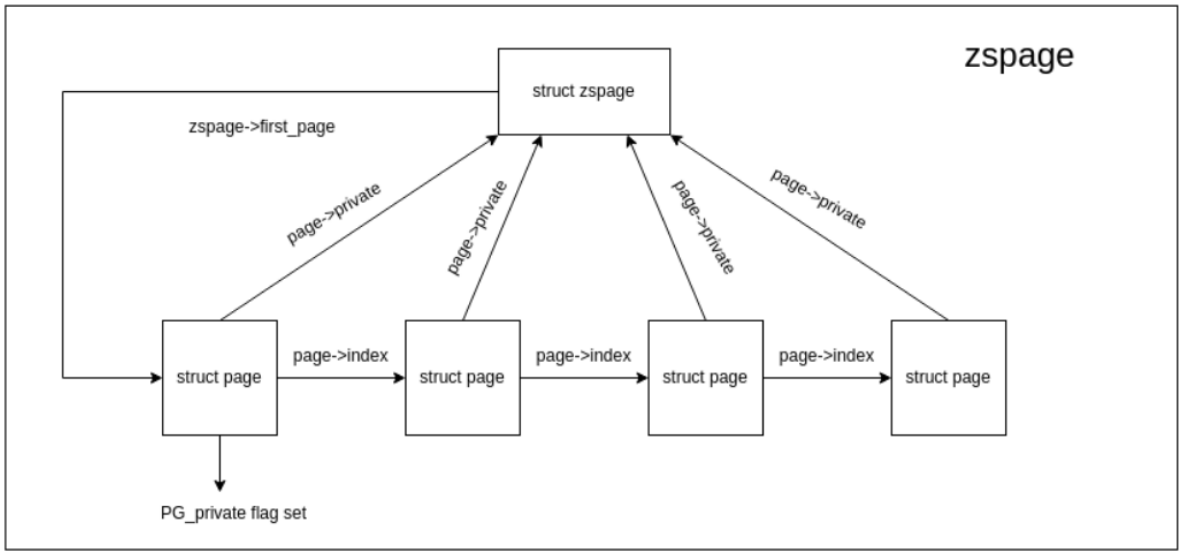

Рисунок 8 – Схематическое описание структуры данных zspage

Структуры `struct page` связаны друг с другом при помощи поля `index`, которое является указателем на начало каждой последующей страницы списка. Для вычисления, к какой именно структуре zspage страница относится, используется поле `private`, являющееся указателем на соответствующую структуру `struct zspage`. У первой странице в списке установлен флаг `PG_private`, а так же структура `struct zspage` имеет указатель на эту страницу. Максимум в списке может быть 4 страницы, минимум – одна.

На рисунке 9 представлено расположение выделяемых объектов в памяти.

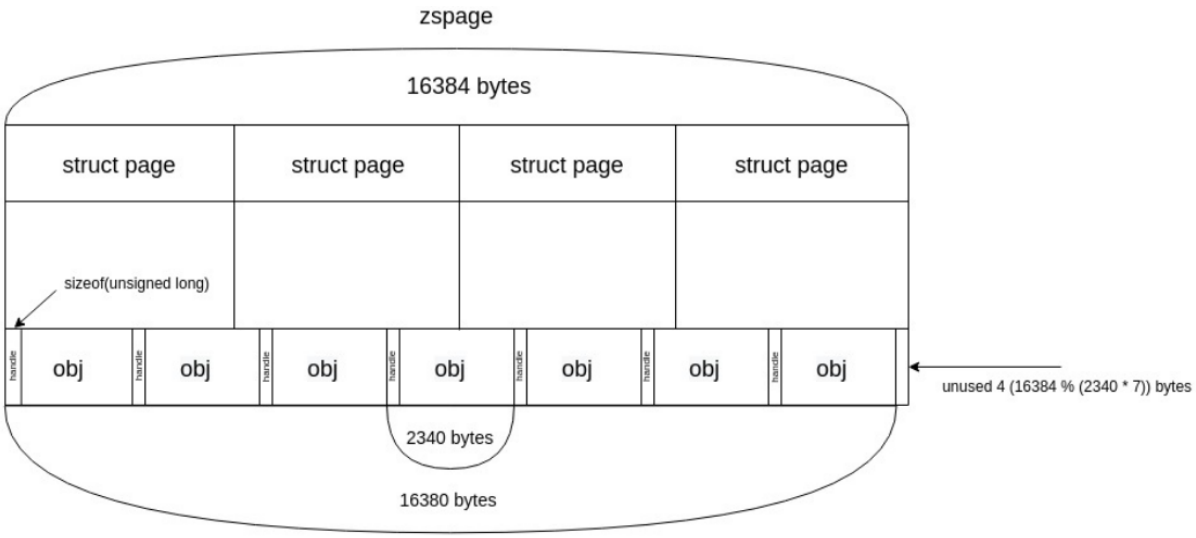

Рисунок 9 – Расположение объектов, хранящихся в zspage, в памяти

Объекты (обозначены именем `obj`), находятся в памяти друг за другом. Перед каждым находится его дескриптор, размер которого соответствует `sizeof(unsigned long)`. Это некоторое целое, безналичное число, которое кодирует положение данного объекта в памяти. В случае, если объект не выделен, дескриптор указывает на следующий свободный объект, тем самым формируя список свободных объектов внутри zspage.

На рисунке 10 представлено описание дескриптора, описывающего запрашиваемые у аллокатора объекты.

Дескриптор представляется из себя целое число, состоящее из `sizeof(unsigned long)` байт. Для простоты изложения, его размер указан в качестве 8 байт (64 бита).

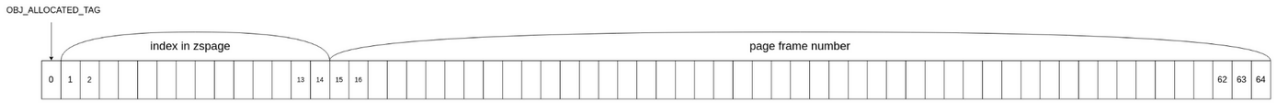

Рисунок 10 – Дескриптор объекта

Первый бит является тегом, позволяющим определить, свободен ли объект. Если первый бит установлен в 0, это значит, что область памяти, находящаяся за дескриптором, не используется. В таком случае, оставшиеся 63 бита являются указателем на следующий свободный участок памяти. В обратном случае (то есть объект уже кем-то используется) биты с 1 по 14 описывают индекс объекта внутри zspage. Оставшиеся биты являются порядковым номером страницы `struct page` в глобальном массиве `mem_map`, который был описан ранее. При выделении объекта, функция `zs_malloc` возвращает данный дескриптор.

Страницы zspage связаны друг с другом с помощью связанного списка. Для каждого размера объекта, в системе существует 4 связанных списка, хранящие страницы zspage:

- `ZS_EMPTY` – список, хранящий страницы zspage в которых все объекты свободны;
- `ZS_FULL` – список zspage, в которых все объекты заняты;
- `ZS_ALMOST_EMPTY` – список zspage, в которых количество занятых объектов не превышает $\frac{3*obj}{4}$, где `obj` – общее количество объектов в странице;
- `ZS_ALMOST_FULL` – список zspage, не попавшими ни в один из вышеперечисленных списков.

На рисунке 11 представлено взаимодействие списков `ZS_EMPTY`, `ZS_FULL`, `ZS_ALMOST_EMPTY`, `ZS_ALMOST_FULL` и страниц zspage.

Для эффективного размещения объектов разного размера в памяти, используется структуры данных называемая size class и zspool, в исходном ко-

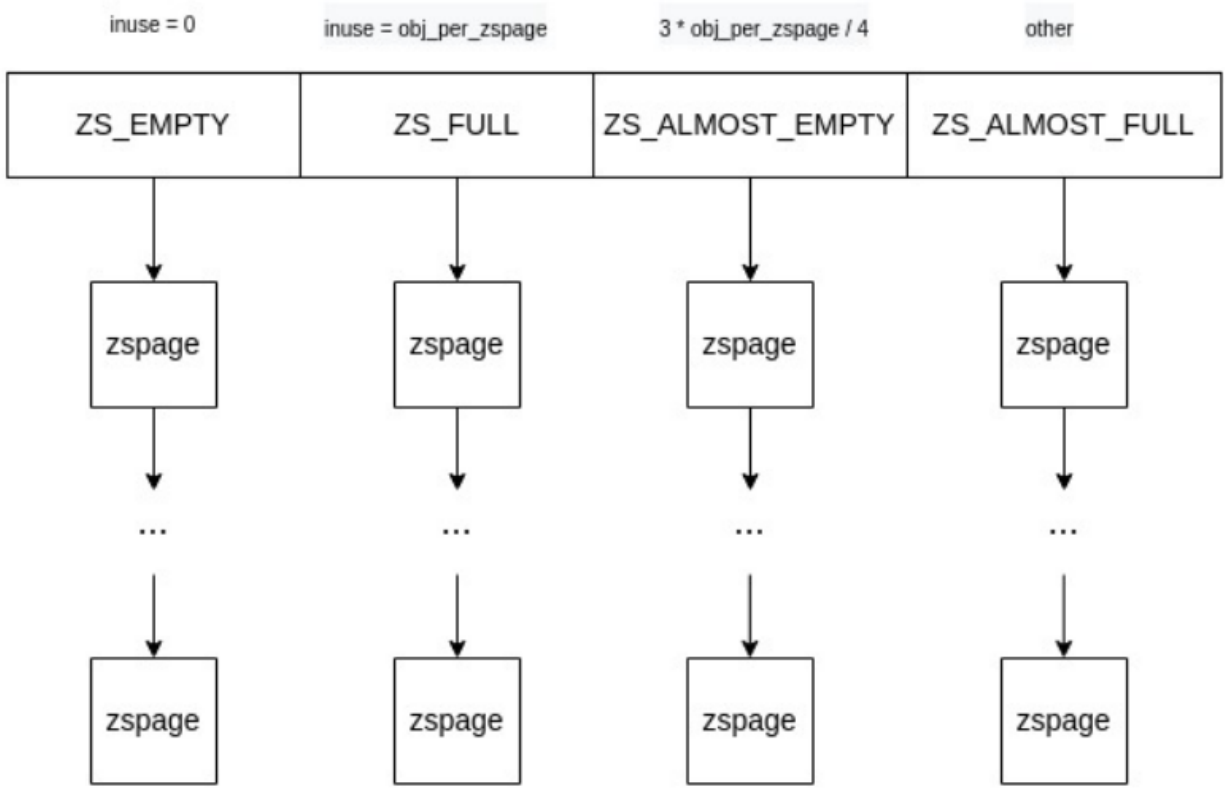

Рисунок 11 – Взаимодействие zspage

де описываемые соответственно структурами `struct size_class` и struct zs_pool (см. листинги 12 – 13).

Листинг 12: Структура `struct size_class`

```
struct size_class {
    spinlock_t lock;
    struct list_head fullness_list[NR_ZS_FULLNESS];
    int size;
    int objs_per_zspage;
    int pages_per_zspage;

    unsigned int index;
    struct zs_size_stat stats;
};
```

Структура данных size_class предназначена для хранения всех объектов размером `size`. Данная структура данных хранит в себе указатели на четыре списка, описанных ранее, которые в свою очередь указывают на страницы zspage. Поле `objs_per_zspage` хранит в себе количество объектов, размещаемых в страницах zspage для данного size class, а в поле `page_per_zspage` хранится количество страниц памяти, используемых внутри zspage.

Листинг 13: Структура `struct zs_pool`

```
struct zs_pool {
    const char *name;

    struct size_class *size_class[ZS_SIZE_CLASSES];
    struct kmem_cache *handle_cachep;
    struct kmem_cache *zspage_cachep;

    atomic_long_t pages_allocated;

    struct zs_pool_stats stats;

    /* protect page/zspage migration */
    rwlock_t migrate_lock;
};
```

zspool – самая «верхняя» структура данных, используемая в алгоритме работы аллокатора zsmalloc. Данная структура хранит в себе массив типа `size_class`, таким образом, каждому пулу принадлежит `ZS_SIZE_CLASSES` соответствующих структур. Суть заключается в том, что каждый size_class отличается от предыдущего размером объектов, которые он хранит. Таким образом, самый size_class хранит объекты размером `PAGE_SIZE`, второй объекты размером `PAGE_SIZE - C`, третий `PAGE_SIZE - 2 * C` и так далее. На

данный момент, $C = 8$, что позволяет создать до 255 size class, при размере страницы 4 Кб. При запросе у аллокатора участка памяти размером n байт, n будет округлено до ближайшего размера size class. Несмотря на то, что размер будет округлен, благодаря большому количеству size class, количество на самом деле неиспользуемых байт будет минимально возможным. Такой подход позволяет максимально эффективно распределять и управлять участками памяти.

На рисунке 12 представлено взаимодействие всех описанных в этом разделе структур данных.

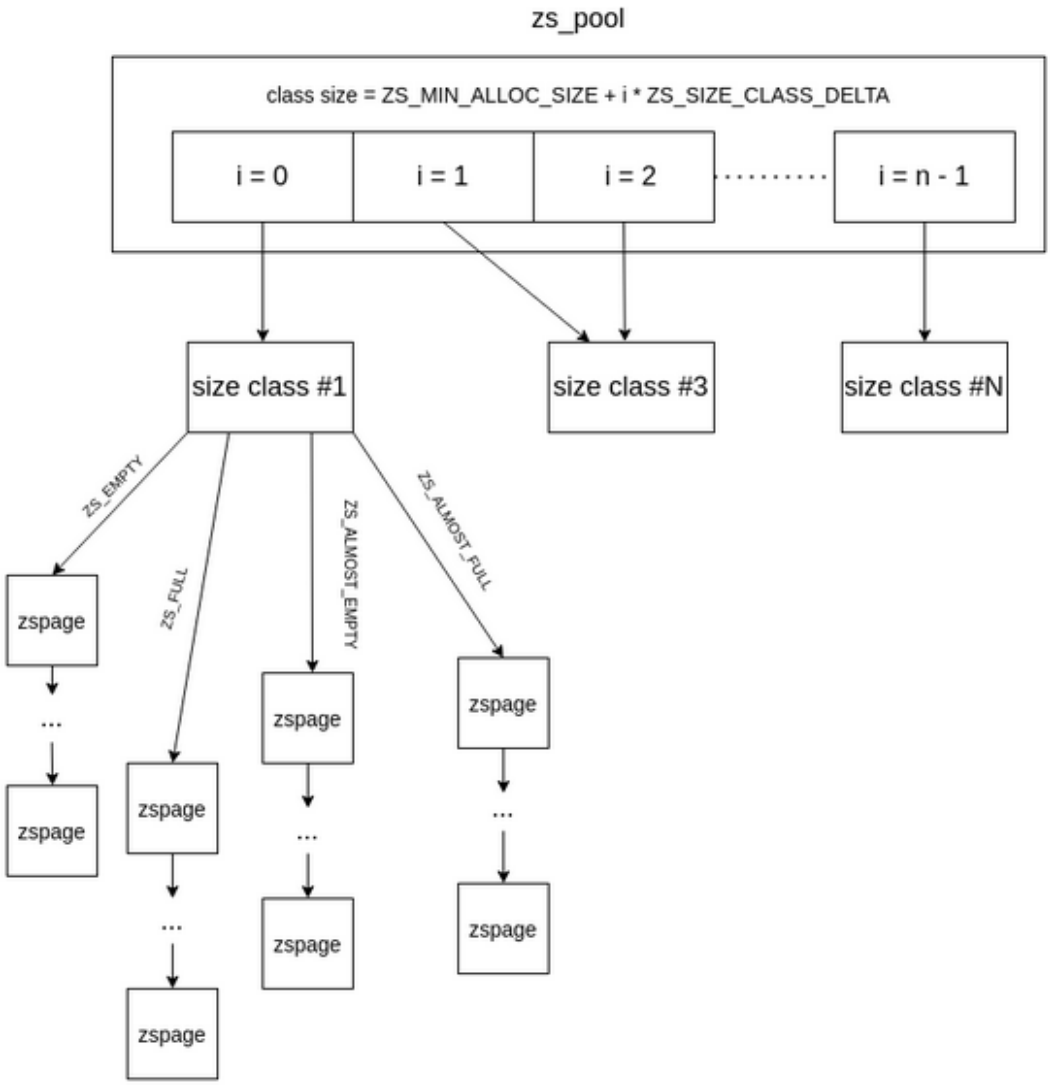

Рисунок 12 – Взаимодействие структур данных аллокатора zsmalloc

Недостатком аллокатора zsmalloc является тот факт, что он предназначен для работы лишь с объектами, размер которых не превышает `PAGE_SIZE` байт.

2.3 Управление несмежными областями памяти

Области в памяти предпочтительно отображать в последовательности физически смежных страниц памяти, обеспечивая эффективное использование кэша и уменьшая среднее время доступа к памяти. Если запросы на области памяти происходят нечасто, или для решаемой задачи не имеет значение физическая смежность страниц, можно использовать схему, основанную на физически несмежных страницах, обращение к которым происходит смежные линейные адреса. Такой подход позволяет избегать внешней фрагментации, но, при этом, заставляет менять таблицу страниц.

В ядре Linux есть специальный участок памяти, зарезервированный для аллокаций такого типа. На рисунке 13 представлен такой участок.

- интервал `PAGE_OFFSET` – `high_memory` – включает в себя отображение первых 896 Мбайт оперативной памяти
- конец области, то есть интервал, начинающийся с `FIXADDR_START`, содержит фиксированно отображенные линейные адреса;
- в интервале `PKMAP_BASE` – `FIXADDR_START` находятся адреса, используемые для постоянного отображения ядром страниц памяти;
- остальные адреса используются под смежные области памяти. Интервалы 8 Мб между `high_memory` и `VMALLOC_START` необходим для обеспечения безопасности и отлавливания обращения памяти по некорректным адресам. По той же самой причине между несмежными областями памяти есть дополнительные интервалы разделом 4 Кб.

Таким образом, для выделения несмежных областей памяти, используются адреса расположенные между `VMALLOC_START` и `VMALLOC_END`.

Для описания каждой несмежной области памяти в ядре Linux используется структура `struct vm_struct`, описание которой представлено в листинге 14. Рассматриваемые дескрипторы объединены в простой односвязный список с помощью поля `next`.

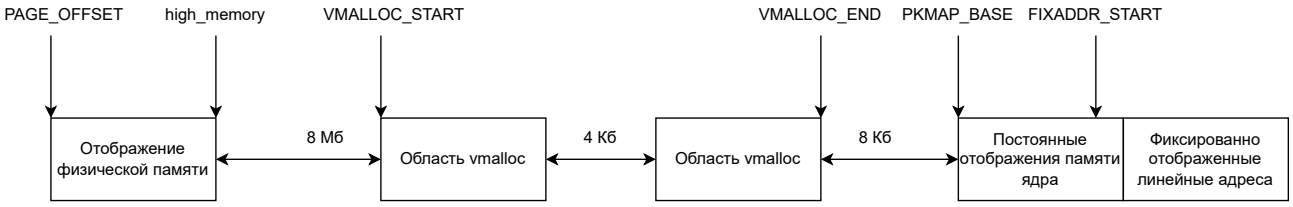

Рисунок 13 – Интервал линейный адресов, начинающийся с `PAGE_OFFSET`

Листинг 14: Структура `struct vm_struct`

```
1  struct vm_struct {
2      struct vm_struct    *next;
3      void            *addr;
4      unsigned long        size;
5      unsigned long        flags;
6      struct page         **pages;
7      unsigned int         nr_pages;
8      phys_addr_t         phys_addr;
9      const void         *caller;
10 };
```

Ниже представлено подробное описание полей:

- `addr` – линейный адрес первой ячейки памяти;
- `size` – размер области + 4 Кб;
- `flags` – флаги, описывающие область памяти;
- `pages` – указатель на массив из `nr_pages` указателей на дескрипторы страниц;
- `nr_pages` – количество страниц, расположенных в области;

Для выделения несмежной области памяти в ядре Linux используются API, представленной в листинге 15.

Листинг 15: API для работы с несмежными областями памяти

```
1  void *vmalloc(unsigned long size);
2
3  void vfree(const void *addr);
4
5  void *vmap(struct page **pages, unsigned int count,
6    unsigned long flags, pgprot_t prot);
7
8  void vunmap(const void *addr);
```

Функция `vmalloc` используется для выделения несмежной области памяти размером `size`, а функция `vfree` соответственно для освобождения такой области памяти. При запросе такой области памяти, функция сначала находит подходящую область памяти, которая описывается структурой `struct vm_struct` и меняет таблицы страниц таким образом, чтобы страницы, на которые указывает эта структура отображались в соответствующие линейные адреса.

Функция `vmap` и `vunmap` работает аналогично, за исключение того, что физические страницы, которые нужно отобразить в соответствующие линейные адреса передаются в качестве параметра.

Данный способ выделения памяти является затратным, так как пользуется изменением страниц, что чревато увеличением времени доступа памяти и отзывчивости всей системы в целом.

2.4 Классификация методов распределения памяти

Для классификации рассмотренных ранее методов были выделены следующие критерии:

- К1 – использование физически смежных адресов памяти
- К2 – произвольный размер запрашиваемой области памяти;
- К3 – максимально возможная утилизация памяти;

- К4 – надежность метода в условиях нехватки памяти;

Результаты классификации рассматриваемых методов описаны в таблице 1.

Таблица 1 – Классификация методов распределения памяти в ядре Linux

Метод	К1	К2	К3	К4
Buddy аллокатор	+	-	+	+
Slab аллокатор	+	+ / -	-	-
Пулы памяти	+	-	+	+
zsmalloc	+	-	+	+
vmalloc	-	+	+	-

В случае Slab аллокатора, при использовании функции `kmalloc`, то есть при обращении к общему кэшу с геометрически распределенными размерами, запрос области памяти возможен, а в случае использования функции `kmem_cache_alloc` – нет.

Таким образом, нельзя сделать вывод о превосходстве какого-либо метода. Каждый метод необходимо использовать для решения конкретной задачи в конкретной ситуации.

ЗАКЛЮЧЕНИЕ

В ходе выполнения научно исследовательский работы была достигнута ее цель – классифицированы методы распределения памяти в ядре Linux.

Для достижения данной цели были решены следующие задачи:

- проведен обзор существующих методов распределения памяти в ядре Linux;
- описать плюсы и недостатки каждого из методов;
- сформулированы критерии классификации;
- классифицированы существующие методы распределения памяти;

Был сделан вывод, что каждый метод необходим для решения конкретной задачи. Таким образом, универсального метода, который был бы применим во всех ситуациях, не существует. Опираясь на классификацию методов, данную в этой научно исследовательской работе, можно делать выводы о применимости алгоритма для той или иной задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. What is Linux? – The Linux Kernel Documentation [Электронный ресурс]. – Режим доступа: https://www.linux.com/what-is-linux/, свободный – (10.11.2022)

2. Ядро Linux. Описание процесса разработки. Третье издание, 2019. Роберт Лав. с. 25 - 36.

3. Using 4KB Page Size for Virtual Memory is Obsolete [Электронный ресурс]. – Режим доступа: https://ieeexplore.ieee.org/document/5211562, свободный – (10.11.2022)

4. Learn the architecture - AArch64 memory management[Электронный ресурс]. – Режим доступа: https://developer.arm.com/documentation/101811/0102/The-Memory-Management-Unit–MMU, свободный – (10.11.2022)

5. Кэш и TLB | IBM [Электронный ресурс]. – Режим доступа: https://www.ibm.com/docs/ru/aix/7.2?topic=implementation-cache-tlbs, свободный – (10.11.2022)