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 НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

«Разработка метода объединения одинаковых объектов для распределителя памяти zsmalloc в ядре Linux»

Студент группы **ИУ7-22М** _____ **А. В. Романов**

(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ **А. А. Оленев**

(Подпись, дата) (И.О. Фамилия)

***2023** г.*

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Существует несколько способов увеличения количества оперативной памяти. Один из способов заключается в физическом увеличении количества планок ОЗУ в системе. Данный способ подразумевает покупку и установку планок ОЗУ, что требует денежных затрат. Кроме физического способа увеличения количества памяти, существуют программные способы увеличения количества ОЗУ, например, сжатие данных. Данный способ требует только вычислительные мощности CPU. Кроме того, к программным способам, можно отнести дедупликацию данных – объединение участков в памяти, содержащих одинаковые данные, в одно целое. Два последних способа можно объединить и получить ещё один наиболее эффективный способ увеличения количества оперативной памяти: дедупликация сжатых данных. Целью данной научно-исследовательской работы является разработка метода объединения одинаковых объектов для распределителя памяти zsmalloc в ядре Linux:

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изложить особенности предложенного метода;
- сформулировать и описать основные этапы метода в виде схем алгоритмов;
- описать структуры данных, используемые в разработанном алгоритме.

1 Алгоритм работы распределителя памяти zsmalloc

zsmalloc – специально спроектированный распределитель памяти (далее – аллокатор) для работы в ситуациях, когда в системе критически мало памяти [1]. Данный аллокатор так же работает с физически непрерывными участками памяти, но максимальный участок памяти, который он может выделить, ограничен размером `PAGE_SIZE`. Данный аллокатор базируется поверх алгоритма buddy-системы [2].

1.1 Функции для работы с распределителем

Аллокатор предоставляет специальное API [3], описанное в листинге 1.

Листинг 1: 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`), которое 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.

1.2 Дескриптор zspage

Единицей диспетчеризации аллокатора является структура данных, называемая zspage, которая описывается соответствующей структурой `struct zspage`, поля которой представлены в листинге 2.

Листинг 2: Структура `struct zspage`

```
struct zspage {
    struct {
        unsigned int huge:HUGE_BITS;
        unsigned int fullness:FULLNESS_BITS;
        unsigned int class:CLASS_BITS + 1;
        unsigned int isolated:ISOLATED_BITS;
        unsigned int magic:MAGIC_VAL_BITS;
    };
    unsigned int inuse;
    unsigned int freeobj;
    struct page *first_page;
    struct list_head list;
    struct zs_pool *pool;
};
```

zspage – это связанный список из объеденных страниц `struct page`, которые указывают друг на друга с помощью поля `index`. zspage – это структура 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 2:

- 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 1 представлена схема взаимодействия структуры `struct zspage` и структур `struct page`.

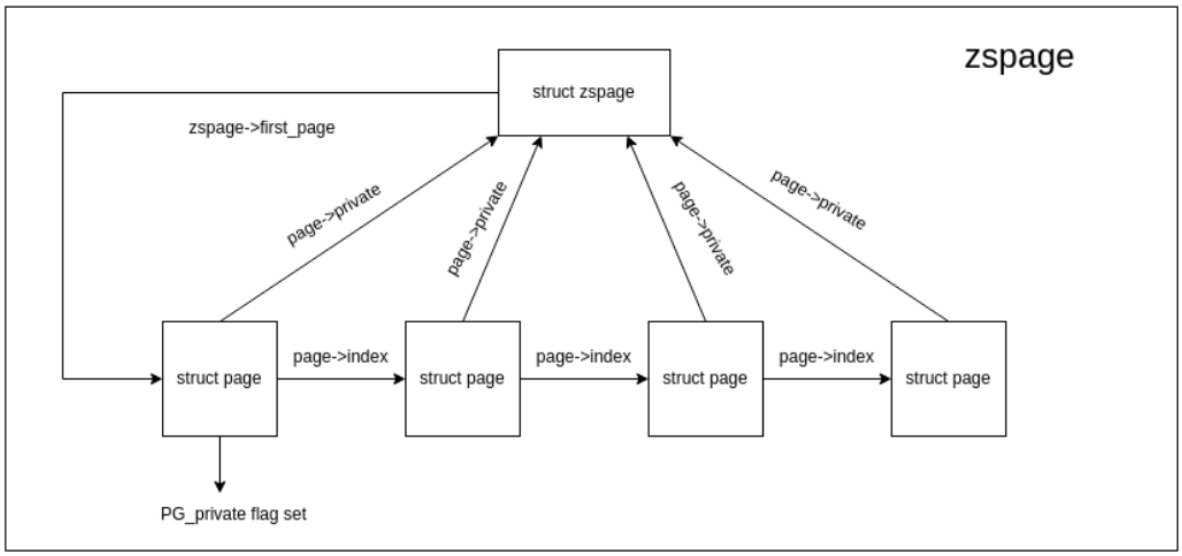

Рисунок 1 – Схематическое описание структуры данных zspage

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.

1.3 Дескриптор объектов

На рисунке 2 представлено расположение выделяемых объектов в памяти. Объекты (обозначены именем `obj`), находятся в памяти друг за другом.

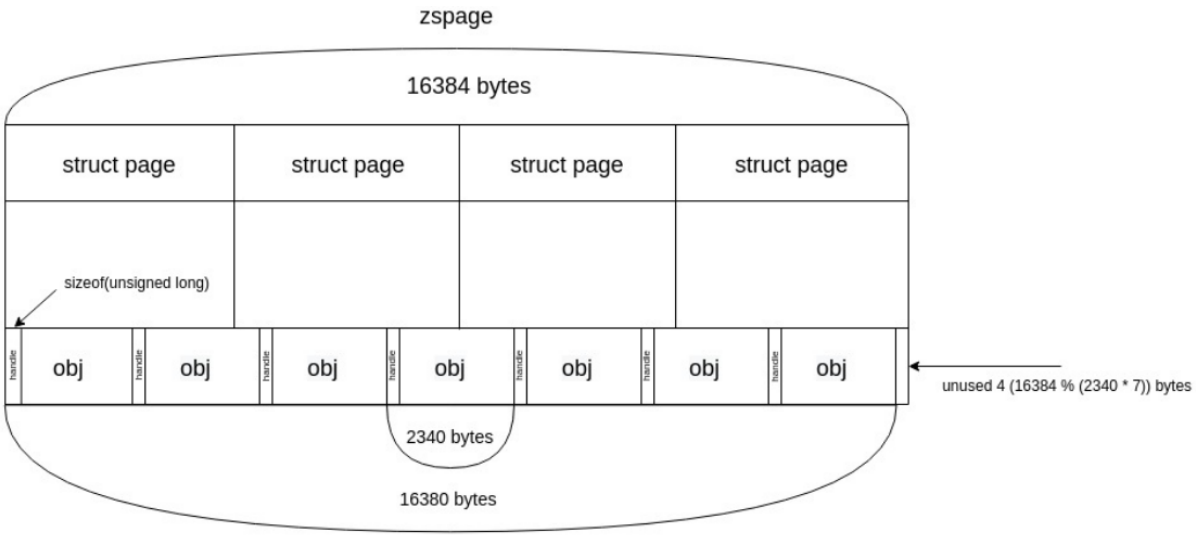

Рисунок 2 – Расположение объектов, хранящихся в zspage, в памяти

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 3 представлено описание дескриптора, описывающего запрашиваемые у аллокатора объекты.

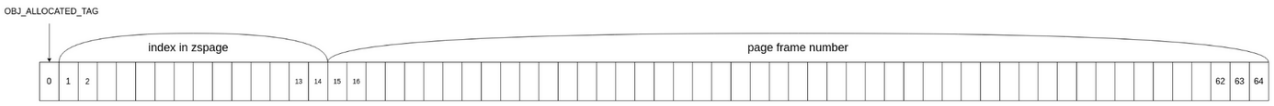

Рисунок 3 – Дескриптор объекта

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

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.

1.4 Хранение страниц zspage

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 4 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 называемые size class и zspool, в исходном коде описываемые соответственно структурами `struct size_class` и struct zs_pool (см. листинги 3 – 4).

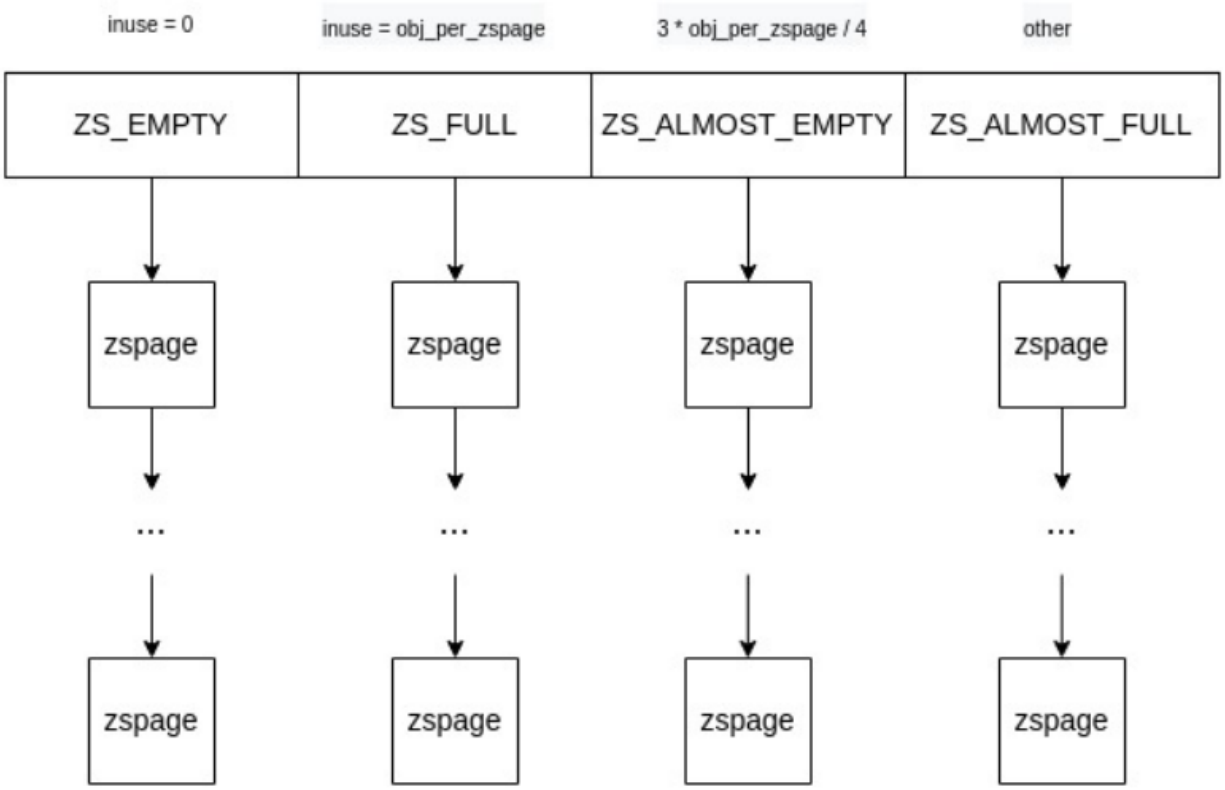

Рисунок 4 – Взаимодействие zspage

Листинг 3: 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 4: 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 5 представлено взаимодействие всех описанных в этом разделе структур данных.

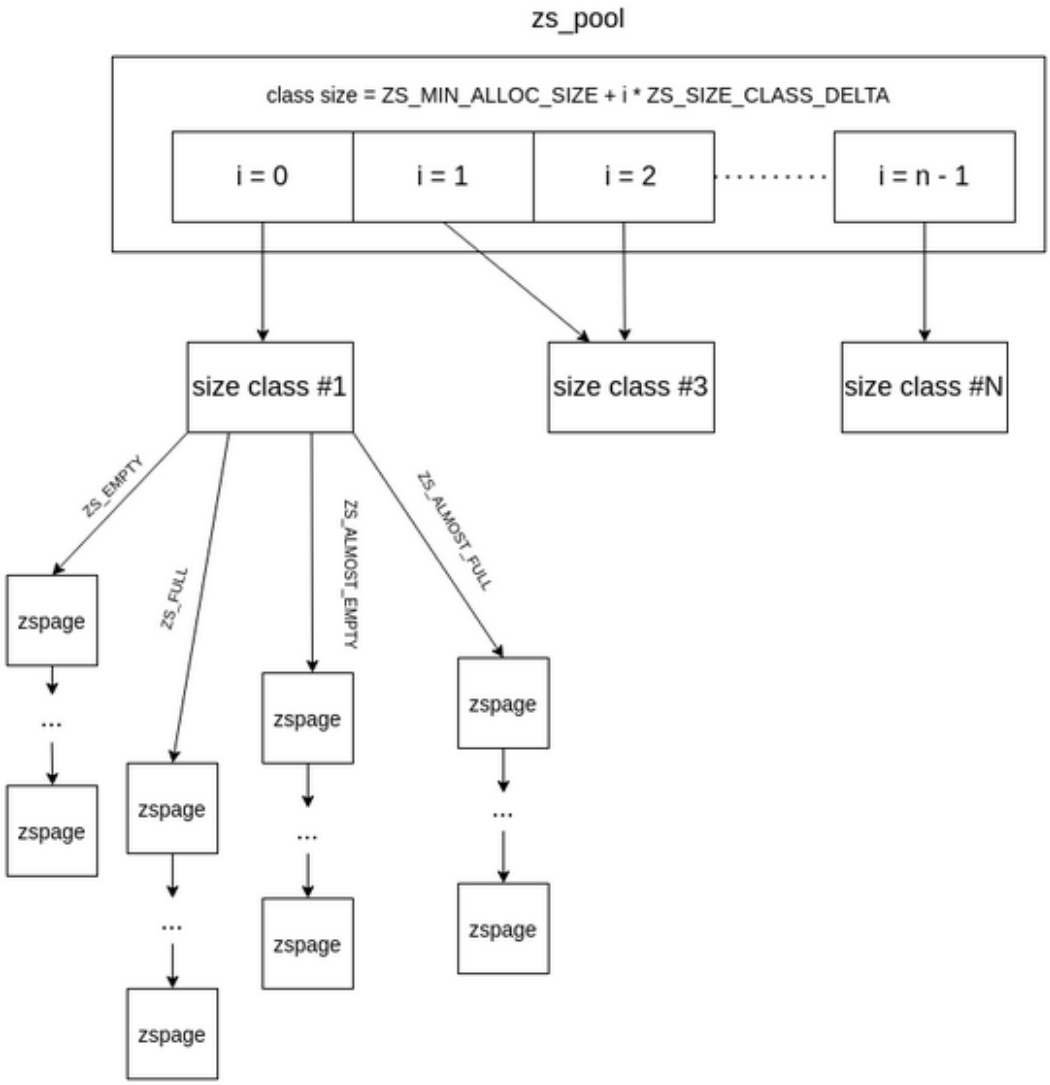

Рисунок 5 – Взаимодействие структур данных аллокатора zsmalloc

2 Алгоритм объединения одинаковых объектов

В данном разделе будет разработан алгоритм объединения одинаковых объектов: изложены особенности метода, представлена схема алгоритма и описаны структуры данных используемые в разработанном алгоритме.

2.1 Особенности метода

Суть метода заключается в сканировании всех объектов, хранящихся в дескрипторе zs_pool, вычисления и сохранения хэш-значения в хэш-таблицу [4]. В случае, если вычисленное хэш-значение, например для объекта i, совпадает с хэш-значением одного из объектов который был обработан ранее, например объекта j, необходимо сделать так, чтобы у объекта i теперь был такой же дескриптор что и объекта j. Участок памяти, на который дескриптор объекта i нужно освободить.

Ниже представлены особенности метода объединения одинаковых объектов, которые необходимо предусмотреть при разработке алгоритма:

- дескриптор типа $size\_class$ хранит объекты одинакового размера. Например, в $size\_class_k$ расположены объекты размером k, а в $size\_class_t$ хранятся объекты размером t. Таким образом, нужно обходить и строить хэш-таблицу не для всех объектов zs_pool, а для каждого $size\_class$ отдельно;
- для того чтобы не освободить область памяти, на которую ссылается дескриптор другого объект, необходимо хранить счётчик ссылок. Для этого можно использовать красно-черное дерево [5]: ключом будет являться дескриптор объекта, а значением – количество ссылок на этот объект;
- стоит учесть, что код ядра Linux может исполняться в нескольких потоках и для того чтобы не повредить структуры данных, необходимо использовать средства синхронизации потоков. Код аллокатора zsmalloc может исполняться в атомарном контексте, что означает что можно использовать только spin-блокировки [6];
- элементы дерева и хэш-таблицы будут выделяться очень часто, причем

их размер всегда будет одинаковым. Таким образом, для выделения этих объектов можно использовать Slab-кэши [7].

2.2 Структуры данных

Для хранения хэш-значений объектов необходимо использовать хэш-таблицу. В данной реализации будет использована хэш-таблица с использованием цепочек. В ядре Linux для реализации таких таблиц существует специальная структура типа `struct hlist_node`.

В листинге 5 представлена структура `struct hash_table`, которая является дескриптором хэш-таблицы.

Листинг 5: Структура `struct hash_table`

```
1  struct hash_table {
2      struct hlist_head *table;
3      struct kmem_cache *cachep;
4      size_t size;
5  };
```

- `table` – массив списков, хранящий элементы таблицы;
- `cachep` – указатель на Slab-кэш, с помощью которого выделяются объекты для таблицы;
- `size` – количество элементов, хранящихся в таблице;

В листинге 6 описана структура `struct obj_hash_node`, которая является ячейкой хэш-таблицы.

Листинг 6: Структура `struct obj_hash_node`

```
1  struct obj_hash_node {
2      unsigned long handle;
3      struct hlist_node next;
4  };
```

- `handle` – дескриптор объекта;
- `next` – следующий элемент в списке (необходимо для решения коллизий).

Для хранения ссылок на объект будет использоваться красно-черное дерево, которое так же уже реализовано в ядре Linux. Необходимо лишь описать дескриптор узла дерева (листинг 7).

Листинг 7: Структура `struct fold_rbtree_node`

```
1  struct fold_rbtree_node {
2      struct rb_node node;
3      unsigned long key;
4      unsigned int cnt;
5  };
```

- `node` – родительская структура ячейки дерева, в которую встраиваются наши данные.
- `key` – ключ, по которому можно идентифицировать узел - совпадает с дескриптором объекта;
- `cnt` – количество ссылок на объект.

2.3 Структура алгоритма

На рисунке 6 представлена IDEF0-диаграмма разрабатываемого алгоритма.

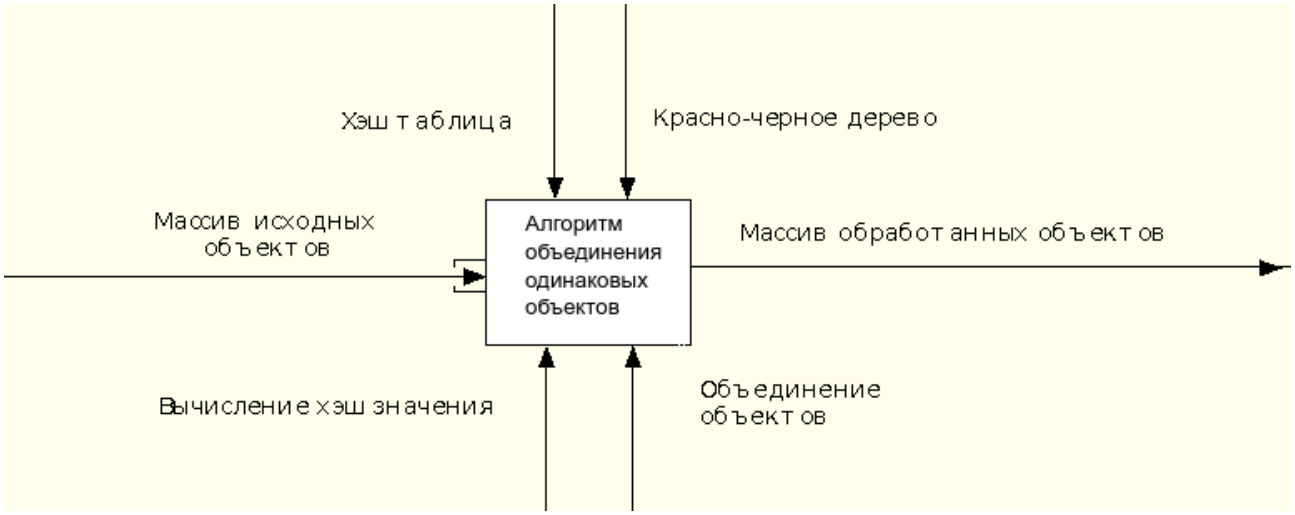

Рисунок 6 – IDEF0-диаграмма разрабатываемого алгоритма

На рисунке 7 представлена детализированная IDEF0-диаграмма разрабатываемого алгоритма.

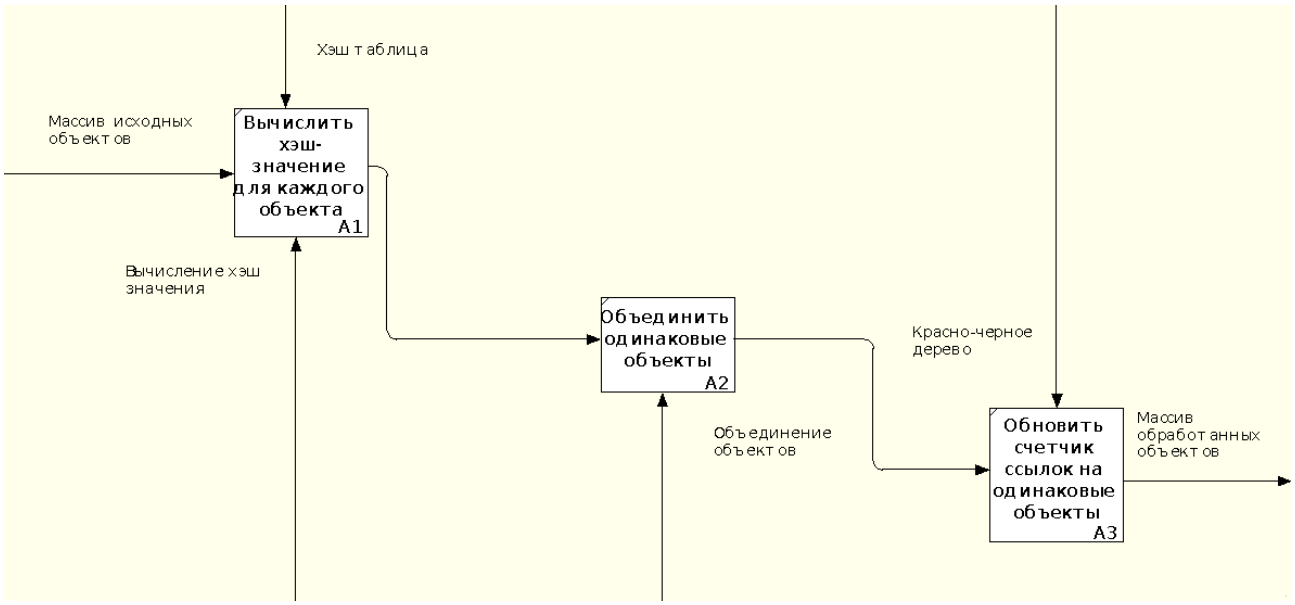

Рисунок 7 – Детализированная IDEF0-диаграмма

2.4 Описание алгоритма

На рисунках 8 - 9 представлена схема разрабатываемого алгоритма.

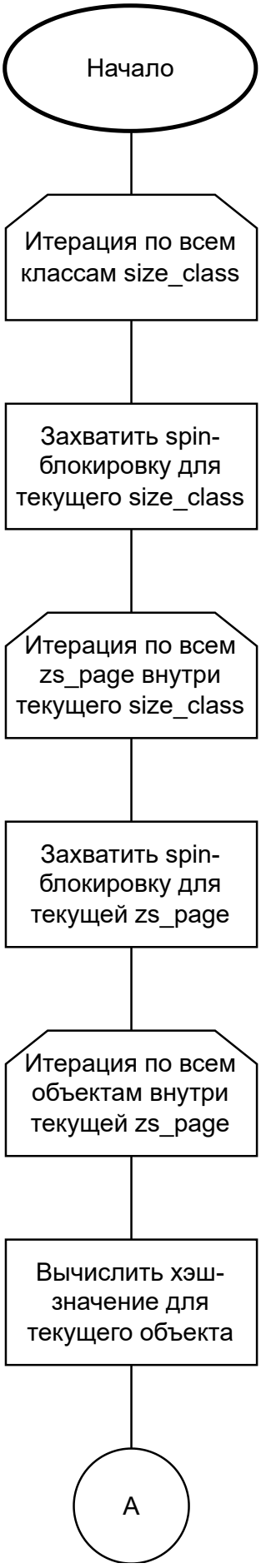

Рисунок 8 – Схема алгоритма объединения одинаковы объектов, часть 1

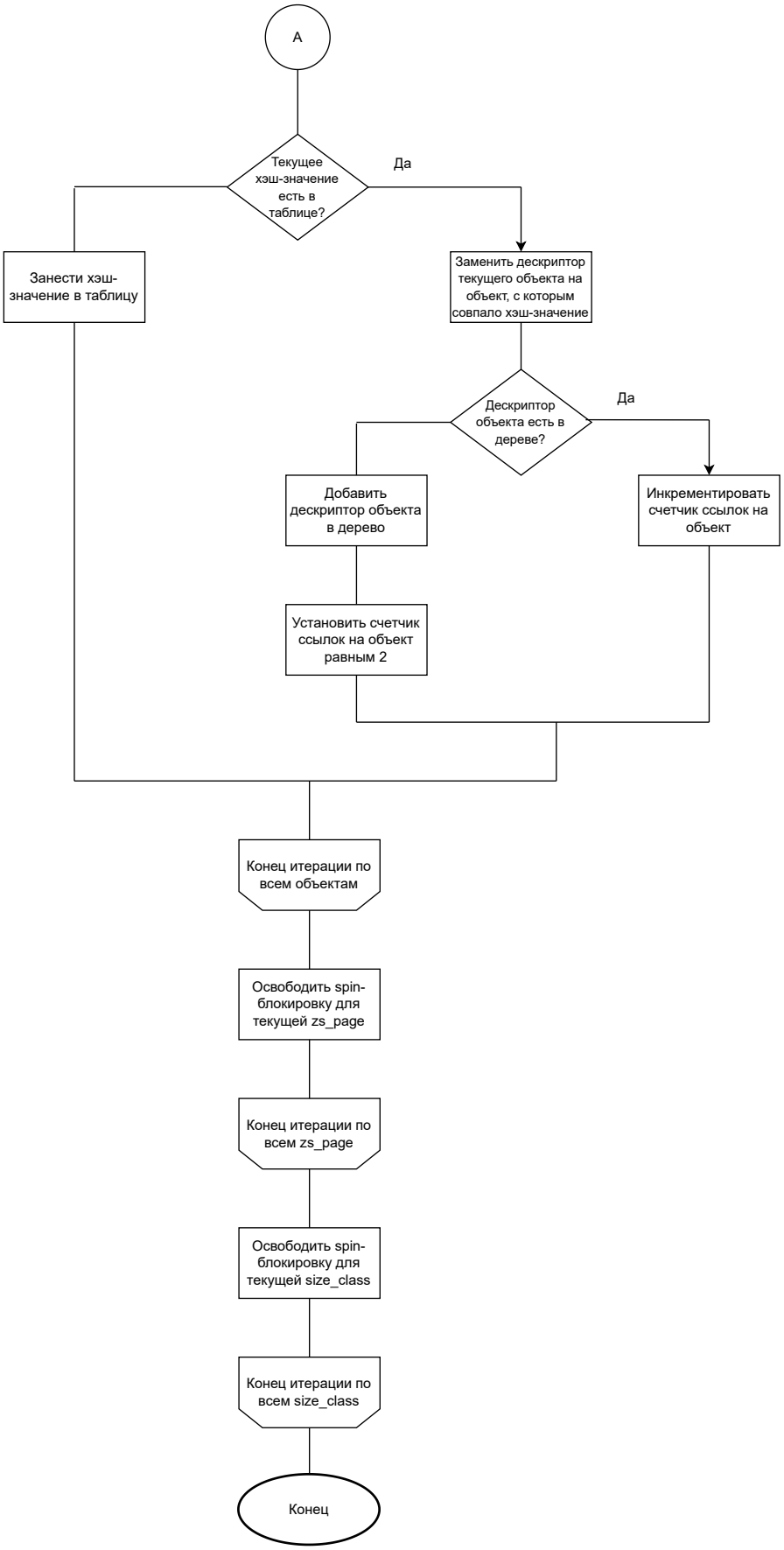

Рисунок 9 – Схема алгоритма объединения одинаковы объектов, часть 2

ЗАКЛЮЧЕНИЕ

В ходе данной работы были выполнены следующие задачи:

- изложены особенности предложенного метода;
- сформулированы и описаны основные этапы метода в виде схем алгоритмов;
- описаны структуры данных, используемые в разработанном алгоритме.

а так же была достигнута её цель – разработан метод объединения одинаковых объектов для распределителя памяти zsmalloc в ядре Linux.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. zsmalloc — The Linux Kernel documentation [Элекронный ресурс]. - Режим доступа: https://www.kernel.org/doc/html/v5.5/vm/zsmalloc.html

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

3. Linux Kernel Source Code — Elixir Bootlin [Элекронный ресурс]. - Режим доступа: https://elixir.bootlin.com/linux/latest/source/mm/zsmalloc.c

4. hash table — IBM [Элекронный ресурс]. - Режим доступа: https://www.ibm.com/docs/en/cics-ts/5.4?topic=overview-hash-table

5. Red-Black Tree — Microsoft [Элекронный ресурс]. - Режим доступа: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/d30e462c-5f8a-435b-9c4c-cc0b9ea89956

6. What is a spin lock? — IBM [Элекронный ресурс]. - Режим доступа: https://www.ibm.com/support/pages/what-spin-lock

7. Slab Allocator — The Linux Kernel documentation [Элекронный ресурс]. - Режим доступа: https://www.kernel.org/doc/gorman/html/understand/understand011