

## **Объединение страниц оперативной памяти, содержащих одинаковые данные, сжатых модулем ядра Linux**

А. В. Романов<sup>1</sup>

romanov.alexey2000@gmail.com

<sup>1</sup>МГТУ им. Н. Э. Баумана, Москва, Россия

### **Аннотация**

Статья посвящена оптимизациям в подсистеме управления памятью в ядре Linux. Кратко описаны главные концепции управления памятью в ядре Linux. Описаны структуры данных и алгоритм работы модуля ядра zram, отвечающего за сжатие страниц оперативной памяти. Разработан метод объединения страниц оперативной памяти содержащие одинаковые данные, которые предварительно были сжаты соответствующим модулем ядра Linux. Проведён анализ результатов работы разработанного алгоритма на различных архитектурах процессора и на разных входных данных.

### **Ключевые слова**

*операционные системы, ядро Linux, управление памятью, zram, сжатие данных, дедупликация данных*

### **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.

## **1 Управление памятью в ядре Linux**

Ядро Linux использует страничную организацию памяти. Суть этого метода заключается в том, что вся физическая память разделена на страницы одинакового

размера, называемые страницами памяти. Чаще всего, размер такой страницы равен 4096 байт, но это число является архитектурно зависимым, и может отличаться от архитектуры к архитектуре. В Linux оно задано константой `PAGE_SIZE`.

Благодаря механизму страничной организации памяти реализуется механизм виртуальной памяти. Виртуальная память – метод управления памятью, при котором физический адрес каждой ячейки памяти автоматически (обычно аппаратно) транслируется в некоторый логический адрес и наоборот. Каждое такое соответствие однозначно. При таком методе управления памятью, программы всегда взаимодействуют с логическими адресами. Благодаря этому, в ядре Linux решаются следующие задачи:

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

Каждая физическая страница памяти в исходном коде ядра Linux описывается структурой `struct page`, которая представлена в листинге 1. Представлены лишь наиболее важные поля структуры – большая часть полей структуры используется в различных ситуациях по разному, поэтому описывать их здесь не имеет смысла.

Листинг 1 — Структура `struct page`

```
1 struct page {  
2     unsigned long flags;  
3     unsigned long private;  
4     atomic_t _refcount;  
5     atomic_t _mapcount;  
6     struct list_head lru;  
7 };
```

Опишем подробно поля, указанные в листинге рассматриваемой структуры:

- `flags` – переменная, содержащая флаги, описывающие данную страницу памяти;
- `private` – специальное поле для хранения различных данных;
- `_refcount` – счетчик ссылок на страницу;
- `_mapcount` – количество записей таблицы страниц, ссылающихся на страницу;
- `lru` – указатель на двунаправленный список давно неиспользуемых страниц.

## 2 Структуры данных и схема работы модуля ядра `zram`

`zram` – модуль ядра Linux, предназначенный для сжатия содержимого страниц оперативной памяти на лету и дальнейшего их сохранения в памяти. При использовании

этого модуля можно увеличить эффективный размер оперативной памяти системы. Так, например, если некоторая система физически ограничена оперативной памятью размером 4 гигабайта, при среднем коэффициент сжатия  $k = \frac{1}{2}$  эффективный размер оперативной памяти увеличивается до 8 гигабайт. Но, стоит отметить: из-за того что сжатие – затратная операция (с точки зрения CPU), рассматриваемый модуль ядра чаще всего используют в ситуациях, когда в системе мало свободной оперативной памяти. В противном случае, при попытках сжатия всей доступной памяти, это бы приводило к уменьшению отзывчивости системы.

Единицей диспетчеризации zram является страница памяти. То есть, данный модуль работает со страницами памяти (со структурой `struct page`), сжимая данные, которые в них хранятся. Модуль zram создает специальное блочное устройство, находящееся в оперативной памяти, которое обрабатывает страницы памяти. Так, например, при попытке записи какого-либо файла в такое блочное устройство, содержимое файла будет разбито на страницы размером `PAGE_SIZE`, которые в последствии будут сжаты и сохранены в оперативной памяти.

Каждая сжатая страница внутри модуля ядра zram описывается структурой `struct zram_table_entry`, которая представлена в листинге 2. Массив таких структур хранится внутри структуры `struct zram`.

Листинг 2 — Структура `struct zram_table_entry`

```
1  struct zram_table_entry {
2      union {
3          unsigned long handle;
4          unsigned long element;
5      };
6      unsigned long flags;
7  }
```

Опишем поля рассматриваемой структуры:

- `handle` – некоторое закодированное число, ссылка на адрес в памяти, где хранятся сжатые данные исходной страницы памяти;
- `element` – если исходная страница состоит из одного и того же элемента, то сохраняется только этот элемент. Так, например, если каждый байт страницы равен нулю, то в поле `element` будет сохранено 0;
- `flags` – флаги, описывающие сжатую страницу памяти. Например, если страница состоит из одинаковых элементов, она будет помечена специальным флагом – то есть у переменной `flags` будет выставлен некоторый бит, отвечающий за этот флаг;

zram использует специально спроектированный аллокатор `zsmalloc`, целью которого является эффективное распределение памяти при маленьком количестве свободной оперативной памяти. В отличие от наиболее используемого подхода, когда пользователь запрашивает у аллокатора участок памяти размера  $n$  и на выходе получает указатель на начало этого участка, `zsmalloc` возвращает некоторое целое без знаковое число, являющееся специальным образом закодированным указателем на необходимую область памяти. Далее, необходимо передать это число в специальную функцию, представленной в интерфейсе аллокатора, и только после этого получить необходимый адрес на запрашиваемый участок памяти. Такая особенность связана с внутренней реализацией `zsmalloc`, и позволяет добиться наиболее эффективного распределения памяти. В листинге 3 представлено API для работы с данным аллокатором.

Листинг 3 — API для работы с `zsmalloc`

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

Рассмотрим подробнее API для работы с аллокатором `zsmalloc`:

- `zs_create_pool` – создать некоторый 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;
- `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 область 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 аллокатором память. После вызова этой функции, обращаться к выделенному участку памяти запрещено.

Таким образом, поле `handle` структуры `struct zram_table_entry` – это закодированный указатель на область памяти, который вернула функция `zs_malloc`, в которой хранятся сжатые данные страницы, доступ к которым можно получить с помощью функции `zs_map_object`.

На рисунке 1 представлена концептуальная схема работы модуля ядра `zram` и его взаимодействие со всей системой.

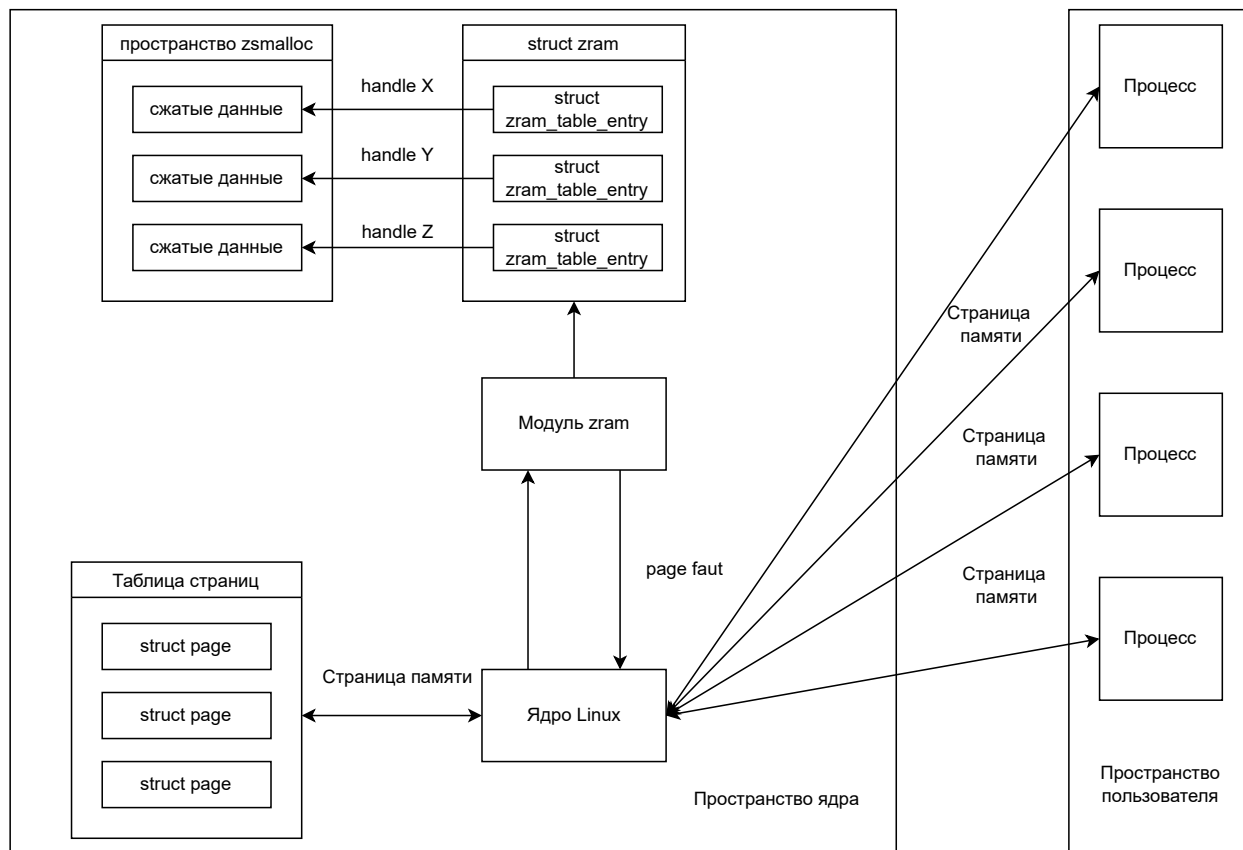


Рисунок 1 — Схема работы модуля ядра `zram`

Приведем алгоритм работы `zram` при попытке записи в блочное устройство:

- содержимое каждой страницы памяти, попавшее в блочное устройство, сжимается и записывается во временный буффер;
- у аллокатора `zsmalloc` запрашивается участок памяти, с помощью функции `zs_malloc`, равный размеру сжатых данных;
- происходит сопоставление закодированного указателя `handle` и выделенной области памяти с помощью функции `zs_map_object`;
- сжатые данные копируются из временного буффера, в область памяти выделенную аллокатором. Временный буффер освобождается.
- заполняется соответствующая ячейка массива структур `zram_table_entry`. В структуре сохраняется указатель на объект – `handle` и в переменную `flags` устанавливаются флаги, описывающие сжатые данные обрабатываемой страницы.

Алгоритм чтения из блочного устройства аналогичен алгоритму записи.

### 3 Алгоритм объединения содержимого страниц оперативной памяти

Рассмотрим ситуацию, представленную на рисунке 2.

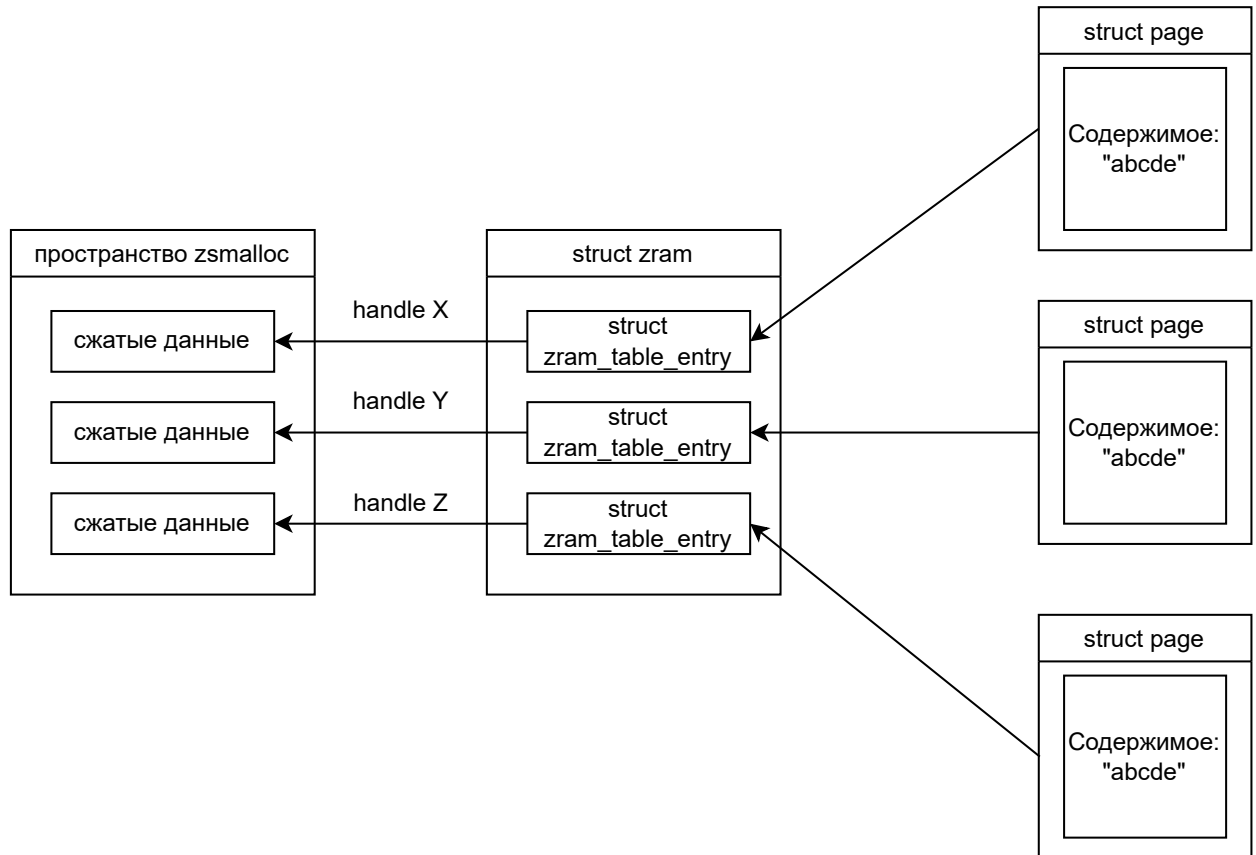


Рисунок 2 — Страницы с одинаковым содержимым

Три страницы с одинаковым содержимым соответствуют трём разным структурам `struct zram_table_entry`, которые в свою очередь хранят закодированный указатель на данные. Сжатые данные, хранящиеся внутри аллокатора `zsmalloc`, дублируют друг друга. В данном примере, при сжатом размере страницы равным  $n$  байт, модуль `zram` использует  $3 * n$  байт памяти, вместо того чтобы использовать  $n$  байт. Этого можно добиться заменив закодированные указатели `handle Y` и `handle Z` на `handle X`, освободить участки памяти внутри `zsmalloc`, на которые они указывают и добавить счётчик ссылок на объект, на который указывает `handle X`. Таким образом, можно сохранить  $2 * n$  байт оперативной памяти. Данная оптимизация представлена на рисунке 3.

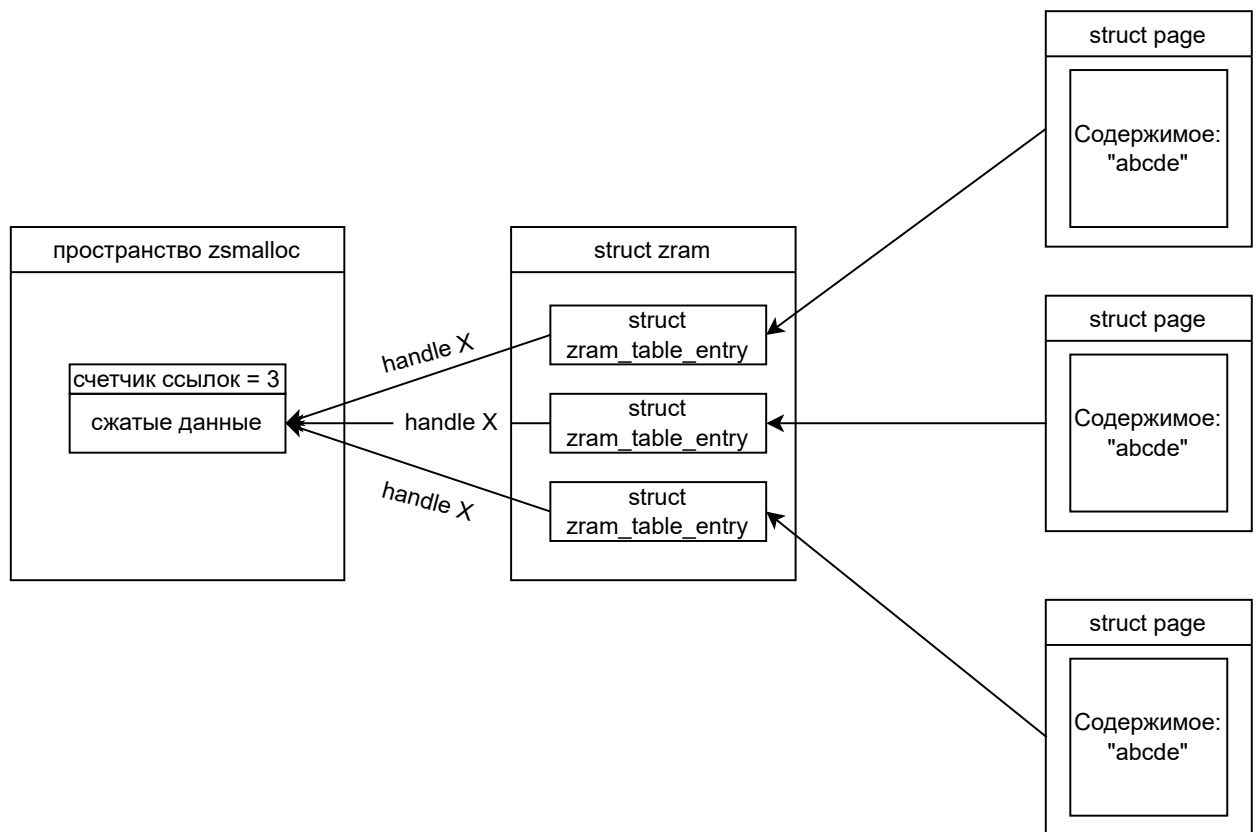


Рисунок 3 — Дедупликация данных

#### 4 Сравнительный анализ результатов работы разработанного алгоритма

#### Заключение

#### Список литературы

**Романов Алексей Васильевич** — студент, МГТУ им. Н. Э. Баумана, кафедра «Программное обеспечение ЭВМ и информационные технологии».