УДК 004.451.3

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

А. В. Романов[1] romanov.alexey2000@gmail.com

[1]МГТУ им. Н. Э. Баумана, Москва, Россия

Аннотация

Статья посвящена оптимизациям в подсистеме управления памяти 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, называющиеся 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 хранится 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 отличии 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 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 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.

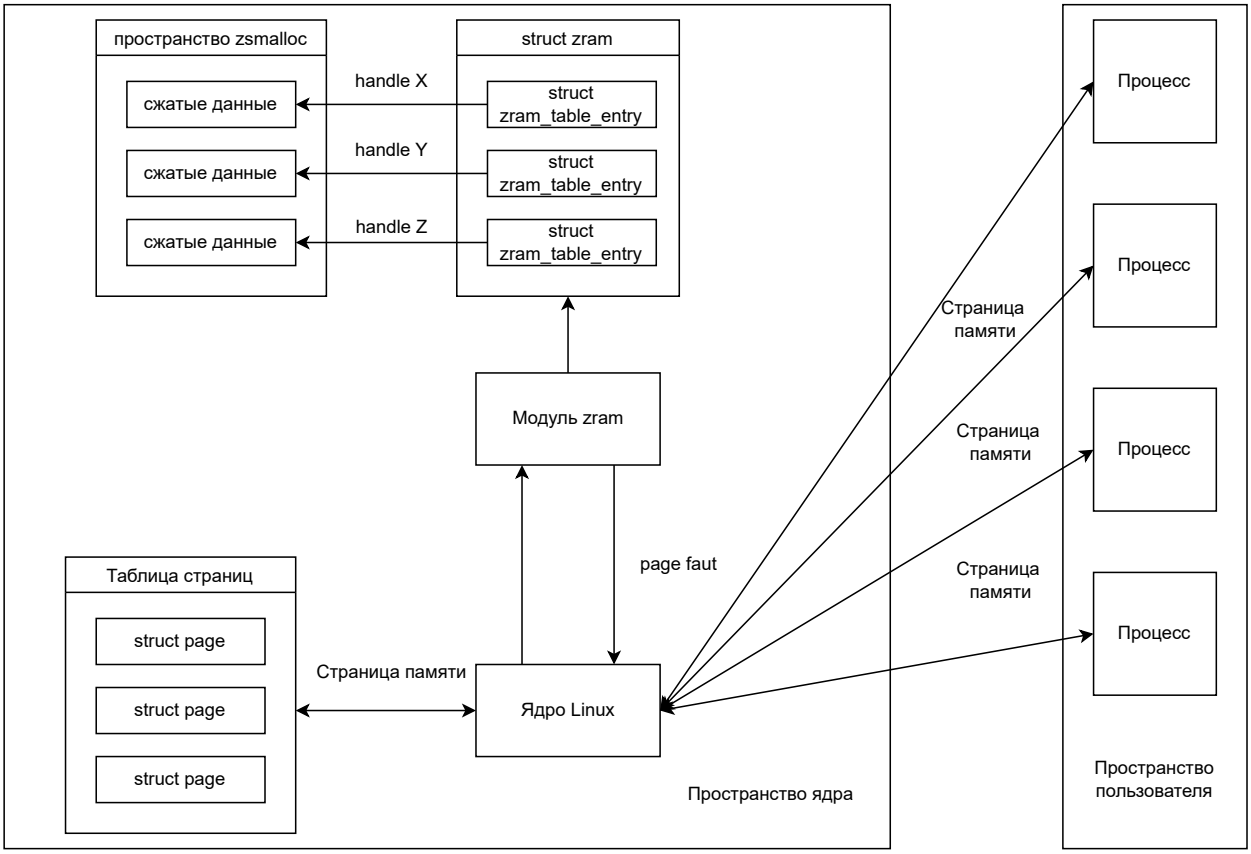

Рисунок 1 — Схема работы модуля ядра zram

Приведем алгоритм работы zram при попытки записи в блочное устройство:

а) содержимое каждой страницы памяти, попавшее в блочное устройство, сжимается и записывается во временный буффер;
б) у аллокатора zsmalloc запрашивается участок памяти, с помощью функции `zs_malloc`, равный размеру сжатых данных;
в) происходит сопоставление закодированного указателя `handle` и выделенной областью памяти с помощью функции `zs_map_object`;
г) сжатые данные копируются из временного буффера, в область памяти выделенную аллокатором. Временный буффер освобождается.
д) 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.

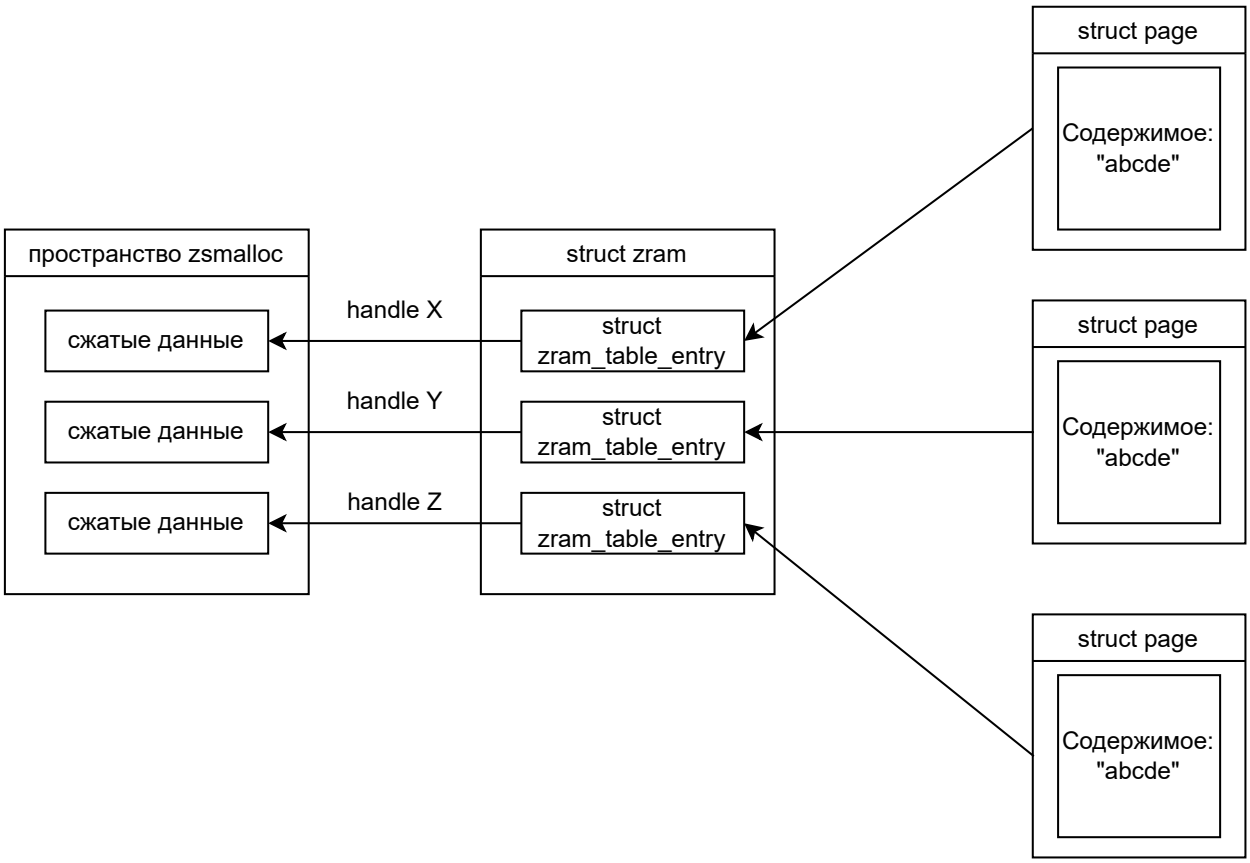

Рисунок 2 — Страницы с одинаковым содержимым

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.

Алгоритм дедупликации сжатых данных можно описать следующим образом;

а) создать хэш-таблицу размерностью n, где n – размер массива структур `struct zram_table_entry`;
б) инициализировать красно-черное дерево, которое будет хранить узлы вида (key, $value$);
в) начать итерироваться по массиву структур `struct zram_table_entry`;
г) с помощью функции `zs_map_object` получить указатель на область памяти, в которой хранятся сжатые данные, на которые указывает очередной элемент

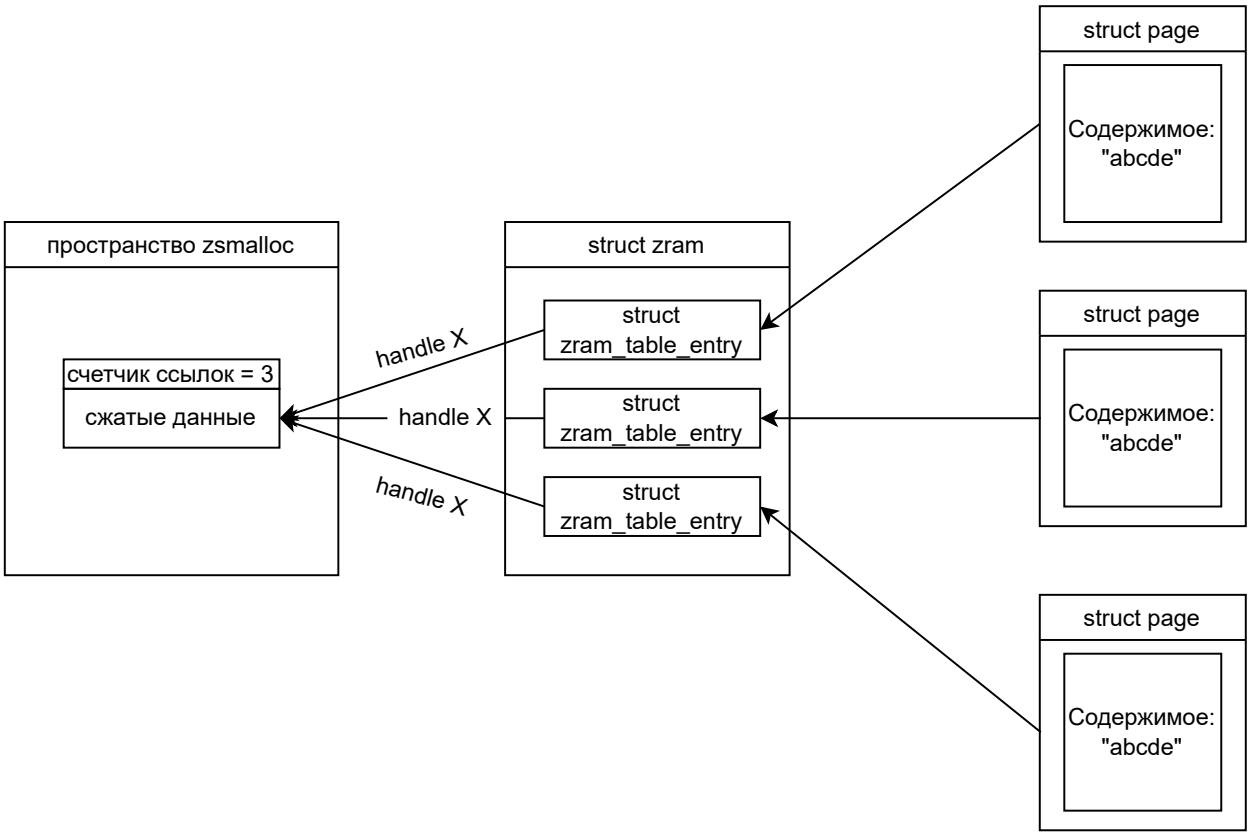

Рисунок 3 — Дедупликация данных

массива с индексом $index$;

д) скопировать данные во временный буффер и вычислить для них хэш-сумму h;

е) проверить, пуста ли ячейка хэш-таблицы с индексом $i = h \ mod \ n$;

ж) если ячейка пуста, то добавить в неё индекс $index$ и перейти к обработке следующего элемента массива;

з) в обратном случае, достать из ячейки хэш-таблицы индекс $index_{hash}$ и получить соответствующий элемент массива структур;

и) повторить шаги г) и д) для этого элемента массива;

к) сравнить две полученные хэш-суммы;

л) в случае их несовпадения, перейти к обработке следующего элемента массива;

м) в обратном случае, с помощью функции `zs_free` освободить участок памяти, на который указывает элемент массива с индексом $index$;

н) заменить указатель $handle$ у элемента массива с индексом $index$ на соответствующий указатель $handle_{hash}$ структуры с индексом $index_{hash}$;

о) проверить наличие в красно-черном дереве узла с ключом $k = handle_{hash}$;

п) если узел уже есть в дереве, инкрементировать значение, хранимое в этом узле (счётчик ссылок на объект $handle_{hash}$);

р) в обратном случае, добавить в дерево узел с ключом $k = handle_{hash}$ и значение $v = 2$ (так как на объект с $handle_{hash}$ на данный момент времени ссылаются два элемента массива);

с) перейти к обработке следующего элемента массива структур.

Красно-черное дерево необходимо для дальнейшей корректной работы системы. Без него, при попытке освобождения области памяти, выделенной аллокатором, невозможно узнать, ссылается ли кто-либо ещё на эту область памяти. Это может привести к непредсказуемым последствиям для всей системы и ядра в целом.

При освобождении одного из элемента массива структур `struct zram_table_entry`, например при прочтении страницы из блочного устройства, необходимо проверить наличие в дереве узла с соответствующем ключом, и, в случае если узел найден, декрементировать значение, хранимое в узле (счётчик ссылок на область памяти). Если счётчик равен нулю (или если узел не найден в дереве), необходимо удалить узел из дерева и с помощью функции `zs_free` освободить область памяти.

Отличительной чертой красно-черного дерева является быстрый поиск и относительно долгое удаление и добавление узлов (из-за того что дерево каждый раз приходиться балансировать). В силу особенности реализации алгоритма, в дереве чаще будет производиться поиск, чем удаление или добавление, поэтому для реализации алгоритма было выбрано именно красно-черное дерево, а, например, не обычное бинарное дерево поиска.

Алгоритм реализован в виде отдельной функции, которую можно вызвать из пространства пользователя. Таким образом, пользователь сам должен определить подходящий момент для соответствующего вызова. Описанный алгоритм может быть переделан таким образом, чтобы хэш-сумма считалась при первичной обработке страницы – то есть в тот момент, когда исходная страница только попадает в блочное устройство в несжатом виде. Но, в таком случае, если система работает с разными данными (мало страниц-дубликатов), zram будет тратить процессорное время на вычисление хеш-сумм в пустую, что ухудшит производительность всей системы. Разработанный подход возлагает ответственность за запуск алгоритма на пространство пользователя, с надеждой что алгоритм будет запущен в тот момент, когда внутри блочного устройства уже находится большее количество страниц-дубликатов.

4 Сравнительный анализ скорости работы разработанного алгоритма

Очевидно, что алгоритм эффективен (по памяти) только в ситуациях, когда в системе имеются дубликаты. Чем их больше, тем сильнее разработанный алгоритм увеличивает эффективный размер оперативной памяти. Поэтому, сравним скорость работы алгоритма при различных состояниях системы на различных устройствах. Тестирование проводилось на трёх устройствах:

а) Система на одном чипе (англ. SoC – System on Chip) с 4-мя ядрами Cortex-A35:
   — Имя устройства: Amlogic SOC S905Y4
   — Процессор: 4 ядра ARM Cortex-A35 @ 2.3 GHz

— Память: 4 ГБ DDR4.

б) SoC с 2-мя ядрами Cortex-A35:
   — Имя устройства: Amlogic SOC A113L
   — Процессор: 2 ядра ARM Cortex-A35 @ 2.3 GHz
   — Память: 128 МБ DDR4.

в) Персональный компьютер с 8-ми ядерным процессором AMD Ryzen 7
   — Имя устройства: персональный компьютер
   — Процессор: AMD Ryzen 7 3700X 8-Core Processor
   — Память: 16 ГБ DDR4.

Алгоритм был протестирован в двух состояниях системы:

— в системе более 25% всех данных – дубликаты;
— дубликаты составляют менее 1% от общего количества данных.

В таблице 1 представлены результаты проведенного тестирования. В ячейках таблицы указано время (в секундах), потраченное на работу алгоритма. Размер исходных данных – 128 МБ.

Таблица 1 — Результаты тестирования разработанного алгоритма на различных устройствах

Устройство	S905Y4	A113L	ПК
25% дубликатов	2.4 сек	2.5 сек	0.45 сек
1% дубликатов	2.2 сек	2.34 сек	0.4 сек

В таблице 2 представлено время (в секундах), потраченное на сжатие данных, попавших в блочное устройство, созданное модулем ядра zram. Размер исходных данных – 128 МБ.

Таблица 2 — Время потраченное на сжатие данных

Устройство	S905Y4	A113L	ПК
25% дубликатов	23.1 сек	40.5 сек	5.2 сек
1% дубликатов	24.1 сек	39.1 сек	5.7 сек

По представленным результатам из таблицы 1 можно сделать вывод о том, что разработанный алгоритм неэффективен когда в системе находится небольшое количество страниц дубликатов – процессорное время тратится на подсчёт хэш-сумм, но из-за того что дубликатов в системе практически нет, количество доступной (эффективной) оперативной памяти, за счёт объединения страниц, не увеличивается. Это обусловлено тем, что объединение страниц не происходит. В итоге, в замен на потраченное процессорное время система не получает ничего.

В обратном случае, когда в системе находится 25% страниц-дубликатов, алгоритм отрабатывает практически с такой же скоростью, как и в случае если в системе таких

страниц менее 1%. Но, в этом случае, система получает выигрыш в виде увеличения объема доступной оперативной памяти.

Кроме того, можно отметить, что скорость работы алгоритма сильно зависит от мощности CPU – чем больше частота процессора, тем быстрее работает алгоритм. При этом, скорость разработанного алгоритма не зависит от количества ядер процессора. Это можно объяснить тем, что алгоритм не является параллельным и выполняется на одном ядре.

Из таблицы 2 можно сделать вывод, что алгоритм работает быстро, как минимум, относительно времени сжатия данных. Алгоритм дедупликации работает быстрее в 10 - 15 раз, чем само сжатие данных. На основе этого, можно сделать вывод о быстродействии алгоритма.

Заключение

Были описаны базовые принципы управления памятью в ядре Linux. Рассмотрены структуры данных модуля ядра zram, описан алгоритм его работы. Разработан алгоритм дедупликации сжатых данных в качестве модификации модуля zram. Проведено тестирование скорости работы алгоритма на различных устройствах.

Разработанный алгоритм оказался относительно неэффективным на системах с маленьким количеством дублирующихся данных. Наоборот, чем больше в системе повторяющихся данных, тем более эффективен разработанный алгоритм. Так же стоит отметить быстродействие алгоритма – объединение страниц происходит в среднем в 10 - 15 раз быстрее, чем сам процесс сжатия страниц.

Разработанный алгоритм вместе с результатами его работы были отправлены разработчикам ядра Linux и модуля zram [**1**]. Результаты работы алгоритма были оценены разработчиками положительно, а сама реализация на данный момент находится на этапе code review, и, возможно, в будущем будет добавлена в основную ветку ядра Linux.

Список литературы

[1] zram: introduce merge identical pages mechanism - Alexey Romanov [Электронный ресурс]. – Режим доступа: https://lore.kernel.org/all/20221121190020.66548-1-avromanov@sberdevices.ru/

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