УДК 004.451.3 DOI: 10.18698/2541-8009-2023-7-923

ОБЪЕДИНЕНИЕ ОДИНАКОВЫХ ДАННЫХ СТРАНИЦ ОПЕРАТИВНОЙ ПАМЯТИ, СЖАТЫХ МОДУЛЕМ ЯДРА LINUX

А.В. Романов rav18u816@student.bmstu.ru

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация

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 алгоритм объединения страниц оперативной памяти, содержащих одинаковые данные, которые предварительно были сжаты соответствующим модулем ядра Linux. Проведен анализ результатов работы разработанного алгоритма на различных архитектурах процессора и на разных входных данных. По результатам исследования сделан вывод о неэффективности разработанного метода на системах с малым количеством одинаковых данных (дублирующие данные составляют не более 1 % всей памяти системы) и, наоборот, об увеличении скорости работы и уменьшении расходования оперативной памяти модуля ядра zram для систем, содержащих повторяющиеся данные в умеренном количестве (дублирующие данные составляют не менее 25 % всей памяти системы).

Ключевые слова

Операционная система, ядро Linux, управление памятью, zram, сжатие данных, дедупликация данных, алгоритмы, структуры данных, блочные устройства, zsmalloc

Поступила в редакцию 23.05.2023

Введение. Существует несколько способов увеличения количества оперативной памяти, один из них 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 объема ОЗУ, например, сжатие данных. Для реализации данного способа требуется только вычислительные мощности CPU [1]. 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 еще один наиболее эффективный способ увеличения объема оперативной памяти — дедупликацию сжатых данных [2].

Управление памятью в ядре Linux. В ядре Linux используется страничная организация 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 [3].

Благодаря механизму страничной организации памяти реализуется механизм виртуальной памяти. Это метод управления памятью, при котором физический адрес каждой ячейки памяти автоматически (обычно на аппаратном уровне) 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 [4]:

– изоляция адресного пространства процессов друг от друга;

– возможность использовать больше оперативной памяти, чем ее установлено в системе;

– устранение необходимости управлять общим адресным пространством.

Каждая физическая страница памяти в исходном коде ядра Linux описывается структурой struct page [5].

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 [6]. При использовании этого модуля можно увеличить эффективный объем оперативной памяти системы. Так, если некоторая система физически ограничена оперативной памятью размером 4 Гб, при среднем коэффициент сжатия k = 1/2 эффективный размер оперативной памяти увеличивается до 8 Гб. Но стоит отметить: поскольку 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. Например, при попытке записи какого-либо файла в такое блочное устройство, содержимое файла будет разбито на страницы размером PAGE_SIZE, которые впоследствии будут сжаты и сохранены в оперативной памяти.

Каждая сжатая страница внутри модуля ядра zram описывается структурой struct zram_table_entry. Массив таких структур хранится внутри структуры struct zram.

В структуре zram используется специально спроектированный аллокатор zsmalloc [7], целью которого является эффективное распределение памяти при маленьком объеме свободной оперативной памяти. В отличие от наиболее распространенного подхода, когда пользователь запрашивает у аллокатора участок памя-

ти размером *n* и на выходе получает указатель на начало этого участка, zsmalloc возвращает некоторое целое беззнаковое 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.

Рассмотрим 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 беззнаковое 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, т. е. 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 рис. 1 представлена концептуальная схема работы модуля ядра zram и показано его взаимодействие со всей системой.

Ниже представлен алгоритм работы zram при попытке записи в блочное устройство:

1) содержимое каждой страницы памяти, попавшее в блочное устройство, сжимается и записывается во временный буфер;

2) у аллокатора zsmalloc запрашивается участок памяти с помощью функции zs_malloc, равный размеру сжатых данных;

3) происходит сопоставление закодированного указателя handle и выделенной областью памяти с помощью функции zs_map_object;

4) сжатые данные копируются из временного буффера, в область памяти выделенную аллокатором. Временный буффер освобождается;

5) 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.

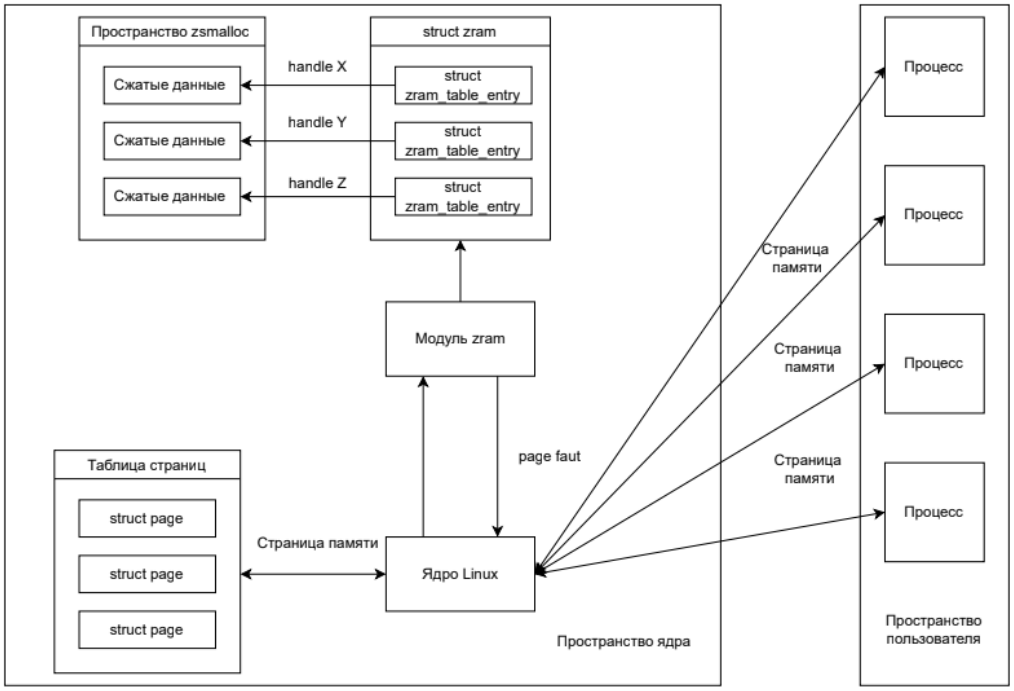

Рис. 1. Схема работы модуля ядра zram

Алгоритм чтения из блочного устройства аналогичен алгоритму записи.

Алгоритм объединения содержимого страниц оперативной памяти. Рассмотрим ситуацию, представленную на рис. 2.

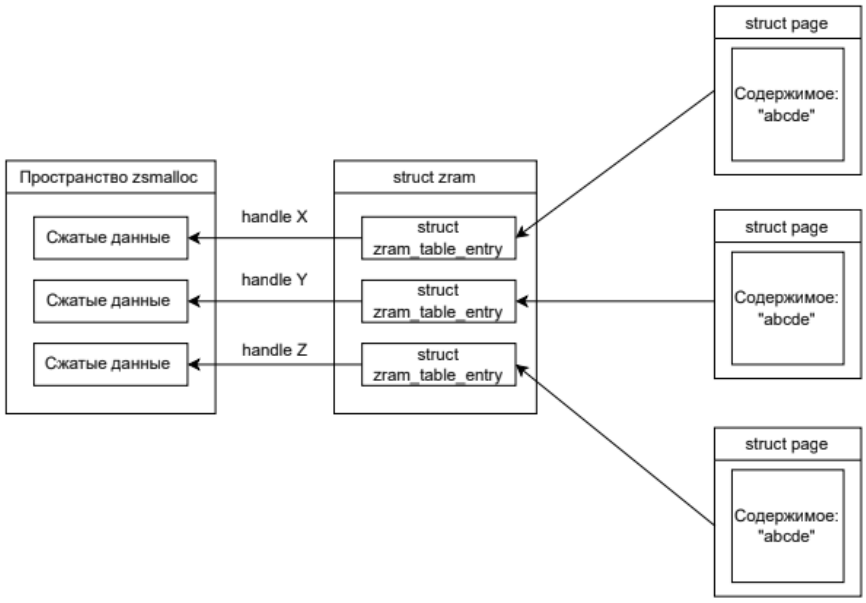

Рис. 2. Страницы с одинаковым содержимым

Три страницы с одинаковым содержимым соответствуют трем 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 *n* байт в модуле zram используется 3*n* байт памяти вместо *n* байт.

Этого можно добиться, заменив закодированные указатели handle Y и handle Z на handle X, освободив участки памяти внутри zsmalloc, на которые они указывают, и добавив счетчик ссылок на объект, на который указывает handle X. Таким образом, можно сохранить 2*n* байт оперативной памяти. Данная оптимизация представлена на рис. 3.

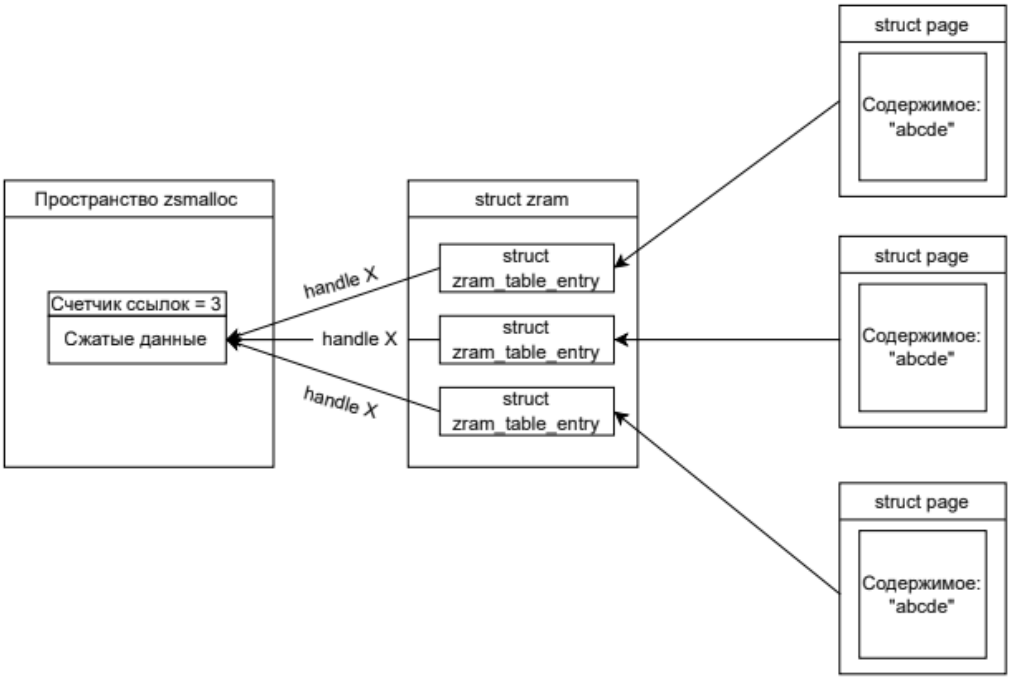

Рис. 3. Дедупликация данных

Алгоритм дедупликации сжатых данных можно описать следующим образом:

1) создание хеш-таблицы размерностью n, где n — размер массива структур struct zram_table_entry;

2) инициализация красно-черного дерева, которое будет хранить узлы вида (k, v);

3) начало интегрирования по массиву структур struct zram_table_entry;

4) получение с помощью функции zs_map_object указателя на область памяти, в которой хранятся сжатые данные, на которые указывает очередной элемент массива с индексом index;

5) копирование данных во временный буфер и вычисление для них хеш-суммы h;

6) проверить, пуста ли ячейка хеш-таблицы с индексом i = hmodn;

7) если ячейка пуста, добавление в нее индекса index и переход к обработке следующего элемента массива;

8) в противном случае извлечение из ячейки хеш-таблицы индекса $index_{hash}$ и получение соответствующего элемента массива структур;

9) повторение шагов 4 и 5 для этого элемента массива;

10) сравнение двух полученных хеш-сумм;

11) в случае их несовпадения переход к обработке следующего элемента массива;

12) в противном случае освобождение с помощью функции zs_free участка памяти, на который указывает элемент массива с индексом index;

13) замена указателя handle у элемента массива с индексом index на соответствующий указатель $handle_{hash}$ структуры с индексом $index_{hash}$;

14) проверка наличия в красно-черном дереве узла с ключом $k = index_{hash}$;

15) при условии что узел уже есть в дереве, инкрементирование значения, хранимого в этом узле (счетчик ссылок на объект $handle_{hash}$);

16) в противном случае добавление в дерево узла с ключом $k = handle_{hash}$ и значения v = 2 (так как на объект с $handle_{hash}$ на данный момент времени ссылаются два элемента массива);

17) переход 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, нет ли других одновременных ссылок на эту область памяти. Это может привести к непредсказуемым последствиям для всей системы и ядра в целом.

При освобождении одного из элементов 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 (счетчик ссылок на область памяти). Если в счетчике присутствует нулевое значение (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 (поскольку дерево каждый раз приходится балансировать) [8]. В силу особенности реализации алгоритма в дереве чаще будет осуществляться 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 алго-

ритм может быть переделан таким образом, чтобы хеш-сумма считалась при первичной обработке страницы — т. е. 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 впустую, что ухудшит производительность всей системы. Разработанный подход возлагает ответственность за запуск алгоритма на пространство пользователя с тем расчетом, что алгоритм будет запущен в тот момент, когда внутри блочного устройства уже находится множество страниц-дубликатов.

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 трех устройствах:

1) система на одном чипе (англ. SoC — System on Chip) с четырьмя ядрами Cortex-A35:

– имя устройства: Amlogic SOC S905Y;

– процессор: 4 ядра ARM Cortex-A35 @ 2.3 GHz;

– память: 4 Гб DDR4;

2) SoC с двумя ядрами Cortex-A35:

– имя устройства: Amlogic SOC A113L;

– процессор: 2 ядра ARM Cortex-A35 @ 2.3 GHz;

– память: 128 Мб DDR4;

3) персональный компьютер (ПК) с восьмиядерным 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:

1) в системе, где более 25 % всех данных — дубликаты;

2) в системе, где дубликаты составляют менее 1 % общего количества данных.

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

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

По данным табл. 1 можно сделать вывод о том, что разработанный алгоритм неэффективен, когда в системе находится небольшое количество страниц дубликатов — процессорное время тратится на подсчет хеш-сумм, но из-за того

что дубликатов в системе практически нет, объем доступной (эффективной) оперативной памяти, за счет объединения страниц, не увеличивается. Это обусловлено тем, что объединение страниц не происходит. В итоге взамен потраченного процессорного времени система не получает ничего.

Таблица 1. **Результаты тестирования разработанного алгоритма на различных устройствах, с**

Устройство	S905Y4	A113L	ПК
25 % дубликатов	2,4	2,50	0,45
1 % дубликатов	2,2	2,34	0,40

Таблица 2. **Время, потраченное на сжатие данных, с**

Устройство	S905Y4	A113L	ПК
25 % дубликатов	23,1	40,5	5,2
1 % дубликатов	24,`	39,1	5,7

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.

По табл. 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 объемом дублирующихся данных. Наоборот, чем больше в системе повторяющихся данных, тем более эффективен разработанный алгоритм. Также стоит отметить быстродействие алгоритма — объединение страниц происходит в среднем в 10–15 раз быстрее, чем сам процесс сжатия страниц.

Реализация алгоритма вместе с результатами его работы были представлены разработчикам ядра Linux и модуля zram [9]. 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.

Литература

[1] Khalid Sayood. *Introduction to Data Compression.* Amsterdam, Morgan Kaufmann, 2001, 680 p.

[2] Dan Feng. Overview of the Data Deduplication. *Data Deduplcation of High Performance Storage System*, 2022, pp. 9–23. http://doi.org/10.1007/978-981-19-0112-6_2

[3] Бовет Д., Чезати М. *Ядро Linux.* Санкт-Петербург, БХВ-Петербург, 2007, 217 с.

[4] Лове Р. *Ядро Linux. Описание процесса разработки*. Москва, Вильямс, 2013, 155 с.

[5] Таненбаум Э., Бос Х. *Современные операционные системы.* Санкт-Петербург, Питер, 2015, 845 с.

[6] *zram: Compressed RAM-based block devices*. URL: https://docs.kernel.org/admin-guide/blockdev/zram.html (accessed April 15, 2023).

[7] *zsmalloc*. URL: https://docs.kernel.org/mm/zsmalloc.html (accessed April 15, 2023).

[8] Бабенко М.А., Левин М.В. *Введение в теорию алгоритмов и структур данных*. Москва, МЦНМО, 2016, 47 с.

[9] Romanov A. *zram: introduce merge identical pages mechanism*. URL: https://lore.kernel.org/all/20221121190020.66548-1-avromanov@sberdevices.ru/ (accessed April 15, 2023).

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

Научный руководитель — Оленев Антон Александрович, старший преподаватель кафедры «Программное обеспечение ЭВМ и информационные технологии», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Ссылку на эту статью просим оформлять следующим образом:
Романов А.В. Объединение одинаковых данных страниц оперативной памяти, сжатых модулем ядра Linux. *Политехнический молодежный журнал*, 2023, № 07 (84). http://dx.doi.org/10.18698/2541-8009-2023-7-923

MERGING IDENTICAL DATA FROM THE MEMORY PAGES COMPRESSED WITH THE LINUX KERNEL MODULE

A.V. Romanov rav18u816@student.bmstu.ru

Bauman Moscow State Technical University, Moscow, Russian Federation

Abstract

The article is devoted to optimizations in the Linux kernel memory management subsystem. Main concepts of memory management in the Linux kernel are briefly described. Data structures and operation algorithm of the zram kernel module responsible for compressing the RAM pages are presented. An algorithm was developed for combining the RAM pages containing the identical data, which were previously compressed by the Linux kernel corresponding module. Results of the developed algorithm operation were analyzed on various processor architectures and on different input data. Based on the study results, it was concluded that the developed method was ineffective for systems with a small amount of identical data (duplicate data was 1 % or less of the total system memory). Conversely, operation speed was increasing with a decrease in the zram kernel module RAM consumption for the systems containing repeated moderate data amount (duplicate data was 25 % or more of the total system memory).

Keywords

Operating systems, Linux kernel, memory management, zram, data compression, data deduplication, algorithms, data structures, block devices, zsmalloc

Received 23.05.2023

References

[1] Khalid Sayood. *Introduction to Data Compression*. Amsterdam, Morgan Kaufmann, 2001, 680 p.

[2] Dan Feng. Overview of the Data Deduplication. *Data Deduplcation of High Performance Storage System*, 2022, pp. 9–23. http://doi.org/10.1007/978-981-19-0112-6_2

[3] Bovet D.P., Cesati M. *Understanding the Linux Kernel*. USA, O'Reilly Media, 2006. (Russ. ed.: Bovet D., Chezati M. *Yadro Linux*. Sankt-Petersburg, BKhV-Peterburg Publ., 2007, 217 p.).

[4] Love R. *Linux Kernel Development*. NJ, Addison-Wesley, 2010, 468 p. (Russ. ed.: Love R. *Yadro Linux. Opisanie protsessa razrabotki*. Moscow, Vil'yams Publ., 2013, 155 p.).

[5] Tanenbaum A.S., Bos H. *Modern Operating Systems*. Pearson Education, 2015, 1138 p. (Russ. ed.: Tanenbaum E., Bos Kh. *Sovremennye operatsionnye sistemy*. Sankt-Petersburg, Piter Publ., 2015, 845 p.).

[6] *zram: Compressed RAM-based block devices*. URL: https://docs.kernel.org/admin-guide/blockdev/zram.html (accessed April 15, 2023).

[7] *zsmalloc*. URL: https://docs.kernel.org/mm/zsmalloc.html (accessed April 15, 2023).

[8] Babenko M.A., Levin M.V. *Vvedenie v teoriyu algoritmov i struktur dannykh* [Introduction to the theory of algorithms and data structures]. Moscow, MTsNMO Publ., 2016, 47 p. (In Russ.).
[9] Romanov A. *zram: introduce merge identical pages mechanism*. URL: https://lore.kernel.org/all/20221121190020.66548-1-avromanov@sberdevices.ru/ (accessed April 15, 2023).

Romanov A.V. — Student, Department of Computer Software and Information Technologies, Bauman Moscow State Technical University, Moscow, Russian Federation.

Scientific advisor — Olenev A.A., Senior Lecturer, Department of Computer Software and Information Technologies, Bauman Moscow State Technical University, Moscow, Russian Federation.

Please cite this article in English as:
Romanov A.V. Merging identical data from the memory pages compressed with the Linux kernel module. *Politekhnicheskiy molodezhnyy zhurnal*, 2023, no. 07 (84). (In Russ.). http://dx.doi.org/10.18698/2541-8009-2023-7-923