author: Пешеходов Андрей aka fresco (filesystems@nm.ru)
released: 15.03.2006
modified: 19.01.2008

Статья была опубликована в журнале "Системный администратор", № 4 (апрель) 2006 г.

Архитектура и реализация reiser4

Многим пользователям ОС Linux нравится не за ее открытость, стабильность и прочие немаловажные характеристики, а прежде всего за гибкость. В Linux, наверное, нет ни одного компонента, которому нельзя было бы подобрать альтернативу. Не являются исключением и файловые системы.

Их насчитывается более двух десятков. Между собой они разнятся не только по дисковой структуре и алгоритмам обработки данных, но и по предоставляемой функциональности. Большую часть составляют так называемые сторонние файловые системы, реализованные для совместимости, например, vfat, ntfs, UFS, и т.д.. “Родных” же для этой ОС файловых систем, обеспечивающих все необходимые функции, до недавнего времени было пять: ext2, ext3, reiserfs, XFS, и JFS.

Сегодня вы познакомитесь с новинкой в этой группе – файловой системой reiser4, спроектированной Гансом Рейзером и его компанией NameSys. Несмотря на название, эта ФС написана с нуля, хотя и унаследовала некоторые особенности своего “идеологического” предшественника – reiserfs.

Помимо традиционных для Linux-ФС функций, reiser4 предоставляет пользователям ряд дополнительных возможностей: прозрачное сжатие и шифрование файлов, полное журналирование данных (реализовано только в ext3), а также практически неограниченную (за счет плагинной архитектуры) расширяемость, т.е. способность приспосабливаться к сколь угодно сложным запросам потребителей. Однако на сегодняшний день отсутствует поддержка direct-IO (начата работа над реализацией), квот и POSIX ACL.

Надеюсь, вы обладаете навыками программирования на языке C и и знакомы с базовыми принципами организации современных файловых систем – без этого понять материал будет трудно.

Примечание: все пути отсчитываются от каталога fs/reiser4/ в дереве исходников ядра Linux.

Плагины

Как говорилось выше, reiser4 основана на плагинах – внутренних программно-обособленных модулях, предоставляющих пользователям возможность максимально адаптировать файловую систему к своим нуждам.

Жестко в драйвер reiser4 зашит только код, занимающийся работой с диском, поддержкой различных абстракций и балансировкой дерева , операции же почти над всеми объектами файловой системы – как внутренними, так и экспортируемыми, как дисковыми, так и in-memory – реализованы в виде плагинов и могут быть расширены дополнительными типами или вовсе заменены. В настоящее время reiser4 не поддерживает динамическую загрузку/выгрузку плагинов (т.е. для подключения нового модуля придется перекомпилировать весь драйвер) однако в будущих версиях файловой системы эта возможность будет реализована.

Строго говоря, reiser4 не имеет жестко определенной ни дисковой, ни алгоритмической структуры, практически любая ее часть может быть легко изменена или дополнена. В этом документе я опишу то, что разработчики называют “format40” -- семейство свойств файловой системы, определенных стандартным набором плагинов от NameSys.

Детали реализации плагинной инфраструктуры будут рассмотрены ниже.

Блоки

Раздел reiser4 представляет собой набор блоков фиксированного размера, пронумерованных последовательно, начиная с нулевого. Максимальное количество блоков на разделе – 2^64.

Раздел начинается с 64 килобайт неиспользуемого пространства, оставленного под загрузчики, дисковые метки и прочие служебные надобности.

Далее следует 2 суперблока – главный и форматный, обрабатываемый

disk_format-плагином. За ними расположен первый bitmap-блок, содержащий битовую карту свободного места. Один бит такой карты соответствует одному блоку файловой системы, если бит установлен – блок занят, если сброшен – свободен. Один bitmap-блок содержит карту для 8*(BLOCK_SIZE-4) блоков файловой системы. Адреса остальных bitmap-блоков считаются по формуле:

```
bmap_block = 8 * (BLOCK_SIZE-4) * N
```

где N – порядковый номер карты, т.е. битовая карта расположена в начале той области дискового пространства, которую она описывает. Это сделано для удобства программы resizefs.reiser4.

Сразу за битовой картой расположены journal header и journal footer блоки, используемые алгоритмом журналирования reiser4, а завершает эту последовательность блок статуса файловой системы, содержащий различные параметры ее состояния.

64 kb unused space	master superblock	format superblock	1-st bitmap block	journal header block	journal footer block	FS status block	data blocks ...

В памяти любой блок представляется объектом *jnode*, описанным в jnode.[ch]. Каждый jnode имеет указатель на дескриптор страницы памяти (struct page* pg), которая содержит данные соответствующего дискового блока, указатель на сами данные (void *data), различные блокировки, счетчики ссылок и биты состояния.

Суперблоки

Структура главного суперблока объявлена в dformat.h и выглядит так:

```
typedef struct reiser4_master_sb {

      char magic[16];         /* Строка "ReIsEr4" */
      __le16 disk_plugin_id;  /* ID форматного плагина */
      __le16 blocksize;       /* Размер блока ФС, в настоящее время
                               * может быть равен только размеру страницы
                               * (4Kb на PC) */
      char uuid[16];    /* Уникальный идентификатор ФС */
      char label[16];   /* Метка ФС */
```

```
     __le64 diskmap;   /* Заготовка на тот случай, если потребуется
                        * изменить положение данных, которые по
                        * умолчанию всегда находятся в определенном
                        * месте */
} reiser4_master_sb;
```

Дисковая структура форматного суперблока обрабатывается плагином format40 и описана в файле plugin/disk_format/disk_format40.h:

```
/* Дисковый суперблок для формата 40, 512 байт в длину */
typedef struct format40_disk_super_block {
     /* Количество блоков в ФС */
     d64 block_count;

     /* Количество свободных блоков */
     d64 free_blocks;

     /* Номер коневого блока дерева ФС */
     d64 root_block;

     /* Наименьший свободный objectid */
     d64 oid;

     /* Количество файлов в ФС */
     d64 file_count;

     /* Сколько раз был сброшен суперблок; пригодится, если в
      * будущем format40 будет иметь несколько суперблоков. */
     d64 flushes;

     /* Уникальный идентификатор ФС */
     d32 mkfs_id;

     /* Строка ReIsEr40FoRmAt”*/
     char magic[16];

     /* Текущая высота дерева ФС */
     d16 tree_height;

     d16 formatting_policy;
     d64 flags;
     char not_used[432];
} format40_disk_super_block;
```

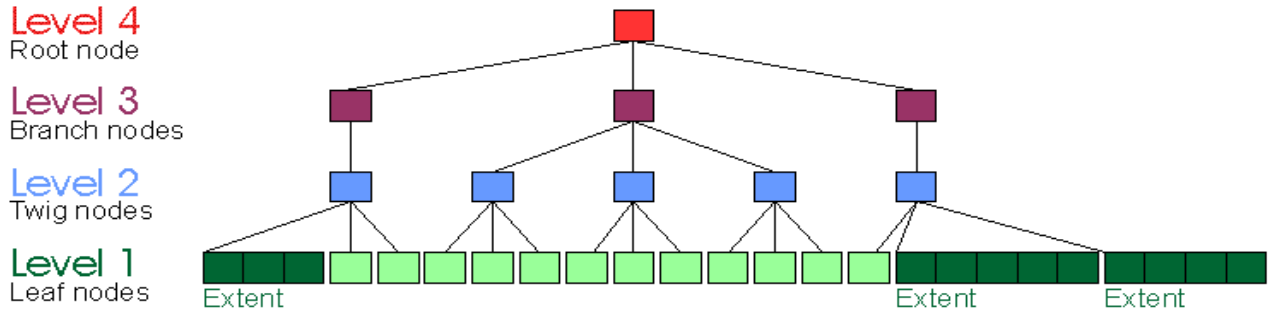
Статус-блок

Формат статус-блока и возможные коды состояния определены в status_flags.h:

```
struct reiser4_status {
      char magic[16];
      d64 status;            /* Текущее состояние ФС */
      d64 extended_status;   /* Некая дополнительная информация о
                              * статусе ФС, например номер сектора, на
                              * котором произошла ошибка ввода-вывода -
                              * если статус определен как
                              * "io error encountered" */
      d64 stacktrace[10];    /* 10 последних вызовов (адреса) */
      char texterror[REISER4_TEXTERROR_LEN];   /* Текст сообщения об ошибке
                                                * (если есть) или нули */
};
```

Дерево файловой системы

Все объекты файловой системы, за исключением суперблока и битовых карт, представлены листьями единственного на всю ФС сбалансированного дерева внешнего поиска – B+ дерева. Такая организация отличает файловые системы семейства reiserfs от аналогов (например, XFS и JFS), имеющих более традиционную структуру, предусматривающую отдельное дерево (а то и не одно – если требуется индексация по нескольким параметрам) для каждой совокупности объектов (например, дерево inodes, экстентов, и др.).



Дерево состоит из внутренних и листовых узлов. Внутренние узлы, в соответствии с заветами классиков, содержат ключи и указатели на потомков (указателей всегда на один больше, чем ключей), листья же, расположенные на самом нижнем уровне дерева, – ключи и данные, организованные в *итемы*.

В связи с тем, что B+ дерево растет вверх, первым уровнем в дереве считается листовой. Над ним расположен twig-уровень (twig – тоненькая веточка), характерный тем, что только на нем существуют extent-итемы (см. ниже), далее идут branch-уровни

(branch – толстая ветвь) и самый верхний – корневой уровень. Минимально возможная высота дерева = 2, то есть корневой узел всегда является внутренним. Такое решение сильно упростило код, а потери дискового пространства были сочтены незначительными.

Узлы и листья дерева представлены в памяти объектами *znode* (см. znode.[ch]), организующимися в древовидную структуру. Znode содержит указатель на jnode, содержащий данные узла, указатели на родителя узла и его соседей в дереве, указатель на плагин, обрабатывающий данный тип узла, блокировки, биты состояния и счетчики ссылок.

Абстракция znode необходима не только для поддержки эффективного кэша узлов – с ее помощью реализуется протокол блокирования объектов файловой системы, улучшающий производительность операций над деревом. Важность этого решения можно оценить на примере reiserfs, не имеющей механизма блокирования элементов дерева. Для синхронизации в ней используется счетчик fs_generetion, позволяющий только установить сам факт изменения дерева. Иногда это приводит к печальным последствиям: если поток, готовящий балансировку, не успевает зафиксировать изменения до того, как дерево было модифицировано другим потоком – вся подготовка выполняется заново.

Во время выполнения балансировки дерево модифицируется по одному уровню за шаг. Во время модификации некоторого уровня файловая система накапливает изменения, которые должны быть распространены на следующий уровень. К примеру, вставка итема (элемента данных, имеющего уникальный ключ) иногда вызывает перемещение итемов между узлами и требует обновления ключа по меньшей мере в общем родителе модифицированных узлов, а в отдельных случаях может потребовать размещения нового узла, указатель на который должен быть вставлен в узел родительского уровня. После того, как все операции на данном уровне дерева выполнены (т.е. итем вставлен, узлы переупакованы), процесс повторяется для накопленных операций на следующем уровне: обновляется существующий или вставляется новый ключ, что также может вызвать переупаковку или размещение нового узла. При этом могут блокироваться и вовлекаться новые узлы, а также выставляться операции для переноса на следующий уровень.

Одно из главных преимуществ по-уровневой балансировки дерева состоит в возможности группировать изменения на родительском уровне и вносить их, в

результате, более эффективно. С деталями реализации кода балансировки можно ознакомиться в файлах carry.[ch].

Ключи

Каждой обособленной части данных или метаданных в reiser4 сопоставляется ключ, являющийся ее уникальным идентификатором. Ключи используются для упорядочивания и поиска данных в дереве ФС. Исходя из того, что плагин “alloc40”, отвечающий за распределение свободного места, старается соблюдать порядок дерева, нельзя не заметить, что политика назначения ключей прямо влияет на производительность файловой системы.

Структура ключа reiser4 определена в key.h

```
union reiser4_key {
      __le64 el[KEY_LAST_INDEX];
      int pad;
};
```

Ключ представляет собой массив из KEY_LAST_INDEX 64-битных чисел, логически раскладываемый на поля. Доступ к конкретному полю осуществляется по двум параметрам: индексу элемента в ключе (reiser4_key_field_index) и смещению поля в элементе (reiser4_key_field_shift). См. key.h:

```
/* Значение каждого элемента этого перечисления есть индекс в
 * массиве reiser4_key->el */
typedef enum {

      /* dirid -objectid  родительского каталога,  расположен в первом
       * элементе, т.н. major "locale" */
      KEY_LOCALITY_INDEX = 0,

      /* Тип итема, расположен в первом элементе, т.н. minor "locale", */
      KEY_TYPE_INDEX = 0,

      /* Существует только в длинных ключах. */
      ON_LARGE_KEY(KEY_ORDERING_INDEX,)

      /* "Объектная связь", второй элемент */
      KEY_BAND_INDEX,

      /* objectid, второй элемент  */
      KEY_OBJECTID_INDEX = KEY_BAND_INDEX,
```

```
      /* Полный objectid, второй элемент  */
      KEY_FULLOID_INDEX = KEY_BAND_INDEX,

      /* Смещение, третий элемент */
      KEY_OFFSET_INDEX,

      /* Хэш имени, в третьем элементе */
      KEY_HASH_INDEX = KEY_OFFSET_INDEX,

      KEY_CACHELINE_END = KEY_OFFSET_INDEX,
      KEY_LAST_INDEX
} reiser4_key_field_index;

/* На сколько бит влево должен быть сдвинут элемент ключа для
 * получения значения конкретного поля */
typedef enum {
      KEY_LOCALITY_SHIFT = 4,
      KEY_TYPE_SHIFT = 0,
      KEY_BAND_SHIFT = 60,
      KEY_OBJECTID_SHIFT = 0,
      KEY_FULLOID_SHIFT = 0,
      KEY_OFFSET_SHIFT = 0,
      KEY_ORDERING_SHIFT = 0,
} reiser4_key_field_shift;
```

Из определения reiser4_key_field_index видно, что ключ в reiser4 может состоять из трех (короткие ключи) или четырех (длинные ключи) 64-битных чисел. Размер ключа определяет макрос REISER4_LARGE_KEY (см. reiser4.h). Если он равен единице, в перечисление, между типом и objectid, добавляется элемент ordering. Драйвер может монтировать только файловые системы с размером ключей, который был установлен во время компиляции. В настоящее время по умолчанию применяются длинные ключи.

Очевидно, что ключ в reiser4 представляет собой нечто большее, чем просто идентификатор – он содержит массу дополнительной информации о типе и положении объекта, которая нигде не дублируется. Для разных типов итемов ключи интерпретируются по разному. Ключи во внутренних узлах дублируют ключи некоторых итемов twig- и листового уровней, они используются только при обходах дерева и не интерпретируются.

Смысл наименований “major” (ID родительского каталога) и “minor” (тип объекта) localities заключается в том, что в дереве объекты физически группируются сначала по dirid (это получается чисто арифметически, т.к. dirid расположен в самых старших битах ключа), а в пределах этой группы – по типу. То есть элементы одного каталога и

одного типа в дереве (а скорее всего и на диске) окажутся соседями.

Текущий алгоритм назначения ключей, реализуемый двумя плагинами – “key_large” (длинные ключи) и “key_short” (короткие ключи), – называется “Plan A” (см. kassign.[ch]).

Короткие ключи

Изначально key_short был основным алгоритмом назначения ключей, однако позднее экономию дискового пространства, достигаемую с его помощью, разработчики сочли незначительной. Был сделан шаг в сторону увеличения производительности – с алгоритмом key_large.

Directory items

60	4	7	1	56	64
dirid	0	F	H	prefix-1	prefix-2/hash
8 bytes		8 bytes			8 bytes

dirid	object id каталога, в котором расположен описываемый файл, + 4 бита типа
F	“волокно” (см. plugin/fibration.c)
H	1, если последние 8 байт содержат хэш 0, если последние 8 байт содержат третий префикс
prefix-1	первые 7 символов имени файла
prefix-2	следующие 8 символов имени файла
hash	хэш оставшейся части имени, не попавшей в prefix-1 и prefix-2

Имена файлов короче 15-ти символов (7+8) полностью помещаются в ключ и именуются короткими. Отличительным признаком таких файлов является бит H=0. Имена других файлов называются длинными, бит H=1, первые 7 символов имени составляют первый префикс, а вторые 8 байт занимает хэш оставшихся символов. Благодаря такой структуре ключа элементы каталога сортируются приблизительно в лексикографическом порядке, а также существенно сокращается количество коллизий (совпадений хэшей у разных имен), однако, принципиально неустранимых при постоянной длине ключа.

Stat-data

60	4	4	64	64
locality id	1	0	objectid	0
8 bytes			8 bytes	8 bytes

locality id object id родительского каталога, 4 бита на тип
objectid object id данного объекта

Extent и tail items

60	4	4	64	64
locality id	4	0	objectid	offset
8 bytes			8 bytes	8 bytes

locality id object id родительского каталога + тип
objectid object id данного объекта
offset логическое смещение от начала файла

Длинные ключи.

В настоящее время используется по умолчанию; позволяет несколько улучшить производительность поиска по дереву в сравнении с key_short.

Directory items

60	4	7	1	56	64	64
dirid	0	F	H	prefix-1	prefix-2	prefix-3/hash
8 bytes				8 bytes	8 bytes	8 bytes

dirid object id каталога, в котором расположен описываемый объект,
+ 4 бита на тип (0 - KEY_FILE_NAME_MINOR, элемент каталога)
F “волокно” (см. plugin/fibration.c)
H 1, если последние 8 байт содержат хэш
0, если последние 8 байт содержат третий префикс
prefix-1 первые 7 символов имени файла

prefix-2 следующие 8 символов имени файла
prefix-3 следующие 8 символов имени файла
hash хэш оставшейся части имени, не попавшей в prefix-1 и prefix-2

Короткими, в данном случае, считаются имена не длиннее 23 символов.

Stat-data items

60	4	64	4	60	64
locality id	1	ordering	0	objectid	0
8 bytes		8 bytes		8 bytes	8 bytes

locality id object id родительского каталога, + 4 бита типа
(1 – KEY_SD_MINOR, stat-data)
ordering копия вторых восьми байт ключа родительского каталога :

```
{
        fibration       :7
        h               :1
        prefix-1        :56
}
```

(см. выше)
objectid object id данного объекта

Такая структура ключа была введена для того, что бы stat-data итемы хранились в каталоге в том же порядке, что и соответствующие им directory entries. Таким образом улучшена производительность вызовов readdir() и stat().

Extent и tail items

60	4	64	4	60	64
locality id	4	ordering	0	objectid	offset
8 bytes		8 bytes		8 bytes	8 bytes

locality id object id родительского каталога, + 4 бита типа
(4 – KEY_BODY_MINOR, тело файла)
ordering см. выше
object id см. выше

offset логическое смещение от начала файла в байтах

Перечисление типов объектов (minor localities) определено в key.h:

```
typedef enum {

      /* Имя файла (элемент каталога) */
      KEY_FILE_NAME_MINOR = 0,

      /* stat-data */
      KEY_SD_MINOR = 1,

      /* Имя атрибута файла */
      KEY_ATTR_NAME_MINOR = 2,

      /* Значение атрибута файла */
      KEY_ATTR_BODY_MINOR = 3,

      /* Тело файла (tail, ctail или экстент) */
      KEY_BODY_MINOR = 4,
} key_minor_locality;
```

Узловые блоки

Каждый дисковый блок, содержащий внутренний или листовой узел дерева, начинается с заголовка узла, состоящего из независимой и форматной частей. Независимый заголовок содержит только идентификатор плагина, который обрабатывает узел данного типа (стандартно – node40-плагин). Форматный заголовок включает служебную информацию, которая интерпретируется соответствующим плагином (количество итемов и свободных байт, различные флаги).

См. plugin/node/node.h

```
/* Независимый заголовок узла */
typedef struct common_node_header {
      __le16 plugin_id;      /* Идентификатор node-плагина, должен
                              * располагаться в самом начале узла */
} common_node_header;
```

См. plugin/node/node40.h

```
/* Node header для узлов формата 40. */
```

```
typedef struct node40_header {
      common_node_header common_header;

      /* Количество итемов. Должен быть первым элементом в заголовке
       * узла */
      d16 nr_items;

      /* Свободных байт в узле */
      d16 free_space;

      /* Смещение начала свободного места */
      d16 free_space_start;

      /* Используются fsck. */
      d32 magic;
      d32 mkfs_id;
      d64 flush_id;

      /* Флаги. Используются fsck и переупаковщиком (repacker) */
      d16 flags;

      /* Уровень узла в дереве. */
      d8 level;

      /* Дополнение */
      d8 pad;
} PACKED node40_header;
```

Внутренние узлы (idef1)

Блок внутреннего узла дерева reiser4 состоит из заголовка узла, массива внутренних итемов и массива заголовков итемов, каждый из которых содержит ключ, pluginid и смещение итема. Оба массива растут к середине.

<table>
<tr><td rowspan="2">Node header</td><td colspan="3">Item heads [0,n]</td><td rowspan="2">Free space</td><td rowspan="2">Internal items [n,0]</td></tr>
<tr><td>key</td><td>pluginid</td><td>item offset</td></tr>
</table>

Формат внутреннего итема описан в plugin/item/internal.h

```
/* Дисковый формат internal item */
typedef struct internal_item_layout {
      reiser4_dblock_nr pointer;
} internal_item_layout;
```

Понятно, что это просто указатель на узел-потомок.

Листья

Листовые узлы расположены на самом нижнем (первом) уровне B+ дерева; поддерживается несколько различных форматов листьев, оптимальных для тех или иных ситуаций.

1. Стандартный формат листовых узлов (leaf1)

<table>
<tr><td rowspan="2">Node header</td><td colspan="3">Item heads [0,n]</td><td rowspan="2">Free space</td><td rowspan="2">Item bodies [n,0]</td></tr>
<tr><td>key</td><td>pluginid</td><td>item offset</td></tr>
</table>

В состав листа входит node header, обязательный для всех узлов дерева, массив заголовков итемов и массив самих итемов (оба растут к середине). Заголовок итема вмещает в себя ключ (формат которого определяется политикой назначения (длинные или короткие ключи) и типом итема, которому соответствует ключ), pluginid и 16-битное смещение начала тела итема. Длина итема вычисляется как разность смещений данного и следующего итемов. Не трудно понять, что длина нулевого итема, расположенного в конце узла, равна:

```
len[0] = node_end –offset[0] +1
```

2. Листья с переменной длиной итемов и ключей (lvar)

<table>
<tr><td rowspan="2">Node header</td><td colspan="3">Item heads [0,n]</td><td rowspan="2">Free space</td><td rowspan="2">Key bodies</td><td rowspan="2">Item bodies [n,0]</td></tr>
<tr><td>key offset</td><td>pluginid triplet</td><td>item offset</td></tr>
</table>

3. Листья со сжатыми ключами (lcomp)

<table>
<tr><td rowspan="2">Node header</td><td colspan="3">Item heads [0,n]</td><td rowspan="2">Free space</td><td rowspan="2">Item bodies [n,0]</td></tr>
<tr><td>key offset</td><td>key inherit</td><td>item offset pair</td></tr>
</table>

Key inherit, однобайтное число, показывает, какая часть префикса совпадает у данного и предыдущего ключей. Соответственно, в данном ключе эта часть не повторяется для экономии места. Таким образом реализовано своеобразное сжатие

ключей.

Итемы

Итем – это обособленная часть данных или метаданных файловой системы, имеющая сопоставленный ей уникальный ключ. Reiser4 поддерживает множество типов итемов, их набор легко может быть расширен написанием дополнительных плагинов.

1. Stat data item

Stat data item содержит метаданные для файлов и каталогов. В чем-то подобен структуре inode других файловых систем, однако, в отличие от классического inode, stat-data не содержит никакой информации о размещении на диске данных описываемого объекта и экспортирует в VFS только атрибуты файла, возвращаемые вызовом stat(2).

В reiser4 структура каждого stat-data набирается из нескольких равноправных компонентов, называемых расширениями. Специальная битовая карта, имеющаяся в каждом stat-data, показывает наличие или отсутствие конкретного расширения.

В соответствии с имеющейся структурой кода, обработчики расширений реализованы в виде плагинов типа REISER4_SD_EXT_PLUGIN_TYPE (о типах плагинов см. ниже). В настоящее время поддерживаются следующие расширения (plugin/item/static_stat.h):

```
/* Перечисление возможных расширений stat-data. */
typedef enum {

     /* Поддержка "легковесных" файлов, атрибуты которых либо
      * наследуются от родительского каталога, либо
      * инициализируются некоторыми разумными значениями */
     LIGHT_WEIGHT_STAT,

     /* Данные, требуемые для реализации вызова stat(2). Формат --
      * reiser4_unix_stat. Если не представлен - файл легковесный */
     UNIX_STAT,

     /* Содержит дополнительный набор 32-битных [amc]time полей для
      * реализации наносекундной точности. Формат - в
      * reiser4_large_time_stat.  Использование этого расширения
      * управляется mount-опцией 32bittimes */
```

```
	LARGE_TIMES_STAT,

	/* Расширение для символических ссылок */
	SYMLINK_STAT,

	/* Если представлено - файл управляется нестандартным плагином
	 * (т.е. плагином, который не может быть вычислен по
	 * mode-битам) */
	PLUGIN_STAT,

	/* Это расширение содержит постоянные (persistent) inode-флаги.
	 * Формат в reiser4_flags_stat. */
	FLAGS_STAT,

	/* Позволяет добавить в stat-data стуктуру capabilities. Сейчас не
	 * используется. */
	CAPABILITIES_STAT,

	/* Содержит размер и public id секретного ключа. Формат в
	 * reiser4_crypto_stat */
	CRYPTO_STAT,

	LAST_SD_EXTENSION,
	LAST_IMPORTANT_SD_EXTENSION = PLUGIN_STAT,
} sd_ext_bits;
```

В том же файле определены структуры соответствующих расширений.

Минимальный stat-data (та самая карта расширений), позволяет поддерживать легковесные файлы, имеется в любом stat-data итеме:

```
	typedef struct reiser4_stat_data_base {
		__le16 extmask;
	} PACKED reiser4_stat_data_base;
```

Расширение для легковесных файлов:

```
	typedef struct reiser4_light_weight_stat {
		__le16 mode;
		__le32 nlink;
		__le64 size;
	} PACKED reiser4_light_weight_stat;
```

Стандартный UNIX-stat, поддерживающий полный набор атрибутов,

возвращаемых вызовом stat(2):

```
typedef struct reiser4_unix_stat {

      /* owner id */
      __le32 uid;

      /* group id */
      __le32 gid;

      /* время последнего доступа */
      __le32 atime;

      /* время последней модификации */
      __le32 mtime;

      /* время последнего изменения */
      __le32 ctime;

      union {
            /* пара [minor,major] для файлов устройств */
            __le64 rdev;

            /* размер в байтах для регулярных файлов */
            __le64 bytes;
      } u;
} PACKED reiser4_unix_stat;
```

Расширение для символических ссылок, содержащее имя, на которое указывает symlink:

```
typedef struct reiser4_symlink_stat {
      char body[0];
} PACKED reiser4_symlink_stat;
```

Контейнер для хранения параметров состояния плагина, самостоятельным расширением не является и входит в reiser4_plugin_stat:

```
typedef struct reiser4_plugin_slot {
      __le16 pset_memb;
      __le16 id;
} PACKED reiser4_plugin_slot;
```

Расширение для файлов с нестандартными плагинами, служит для хранения

параметров состояния требуемого количества плагинов:

```
typedef struct reiser4_plugin_stat {
      __le16 plugins_no;     /* Количество дополнительных
                              * плагинов, ассоциированных с
                              * объектом */

      reiser4_plugin_slot slot[0];
} PACKED reiser4_plugin_stat;
```

Расширение для inode-флагов. В настоящее время это просто 32-битная маска, которая, при необходимости, может быть заменена на маску переменной длины:

```
typedef struct reiser4_flags_stat {
      __le32 flags;
} PACKED reiser4_flags_stat;
```

Расширение для capabilities (сейчас не используется):

```
typedef struct reiser4_capabilities_stat {
      __le32 effective;
      __le32 permitted;
} PACKED reiser4_capabilities_stat;
```

Расширение для хранения размера логического кластера (атрибут cryptcompress-объектов). На самом деле, хранится не сам размер, а его двоичный логарифм, так что размер находится как cluster_size = 1 << cluster_shift.

```
typedef struct reiser4_cluster_stat {
      d8 cluster_shift;
} PACKED reiser4_cluster_stat;
```

Расширение для атрибутов зашифрованных объектов:

```
typedef struct reiser4_crypto_stat {
      d16   keysize;    /* размер секретного ключа в битах */
      d8    keyid[0];   /* ID секретного ключа */
} PACKED reiser4_crypto_stat;
```

Расширение точного времени:

```
typedef struct reiser4_large_times_stat {
      d32 atime;
      d32 mtime;
      d32 ctime;
} PACKED reiser4_large_times_stat;
```

Расширение для плагина single directory entry (сейчас не используется):

```
typedef struct sd_stat {
      int dirs;
      int files;
      int others;
} sd_stat;
```

Tail item

Tail-итемы содержат сырые данные файлов (это либо целые маленькие файлы, либо “хвосты” больших) и не имеют форматной структуры.

Extent items

Экстенты, в терминах reiser4, – непрерывные участки дискового пространства, дескрипторы которых, составляющие extent-итемы, содержат номер стартового блока участка и его длину. Структура экстента определена в plugin/item/extent.h:

```
typedef struct {
      reiser4_dblock_nr start;
      reiser4_dblock_nr width;
} reiser4_extent;
```

В reiser4 экстенты используются для отслеживания только выделенного дискового пространства (в отличие от других ФС, где массивы или деревья дескрипторов свободных участков заменяют битовые карты занятости блоков) и ссылаются на участки, содержащие данные файлов. В ключах экстентов,

принадлежащих одному файлу, равны все поля, за исключением offset (смещение “куска”, описываемого данным экстентом, от начала файла).

Экстент может находиться в одном из следующих состояний (plugin/item/extent.h):

```
typedef enum {
      HOLE_EXTENT,
      UNALLOCATED_EXTENT,
      ALLOCATED_EXTENT
} extent_state;
```

HOLE_EXTENT Экстент представляет собой “дыру” в файле. “Дыра” может образоваться, например, после вызовов:

creat();

truncate(4096);

в результате чего файл длиной 4 kb окажется заполнен неинициализированными данными, выделять под которые дисковое пространство бессмысленно (хотя XFS, к примеру, не только выделяет, но и перезаписывает его нулями). При попытке чтения данных hole-экстента файловая система динамически “сгенерирует” необходимое количество нулевых байт и передаст их пользователю. Дескриптор такого экстента содержит только его длину и никуда не указывает.

UNALLOCATED_EXTENT

“Виртуальный” экстент, являющийся плодом политики отложенного размещения. Появляется в результате добавления данных в файл или при заполнения “дыр”. Существует только в памяти и при сбросе, получив от flush-алгоритма реальный дисковый адрес, превращается в нормальный экстент.

ALLOCATED_EXTEN

Обыкновенный экстент.

Ctail item

Помимо классической реализации регулярных файлов (unix-file плагин), reiser4 предлагает и такую, при которой данные файла хранятся на диске в сжатом и (или) зашифрованном виде (transparent compression/encryption). Ответственным за эту реализацию является cryptcompress-плагин. Основная его идея состоит в том, чтобы выполнять сжатие и шифрование непосредственно перед сбросом кэшированных данных на диск, экономя при этом процессорные ресурсы в том случае, когда одни и те же данные, находящиеся в памяти, многократно модифицируются одним или несколькими потоками. Кроме того, на современных машинах, оснащенных быстрыми CPU и большими объемами оперативной памяти, сжатие данных само по себе не ухудшает, а наоборот – увеличивает производительность файловой системы, т.к. преобразование данных выполняется сравнительно быстро, а объем дискового трафика существенно сокращается.

Каждое преобразование данных (сжатие, шифрование и т.д.) осуществляется некоторым алгоритмом, который присутствует в reiser4 в виде соответствующего плагина (т.н. transform-плагин). Нельзя не отметить пользу плагинной архитектуры, при которой поддержка любого желаемого алгоритма компрессии или шифрования сводится всего лишь к написанию и стандартному добавлению плагина соответствующего типа. В настоящее время доступны transform-плагины для сжатия алгоритмами gzip1 и lzo1, шифровать по идее можно любым блочным алгоритмом, поддерживаемым crypto-API Linux-ядра, а также aes_ecb, однако поддержка крипто-плагинов до ума еще не доведена.

Cryptocompress-плагин разбивает каждый файл на логические кластеры определенного размера. Этот размер является атрибутом данного файла и должен быть назначен перед его созданием. Каждый логический кластер отображается в память на соответствующий страничный кластер, который, в свою очередь, представлен в сбалансированном дереве т.н. дисковым кластером. Сжатые данные такого файла хранятся на диске в виде “фрагментов”, реализованных в reiser4 как итемы специального типа (собственно ctail-итемы), что существенно упрощает произвольный доступ к данным. Каждый логический кластер сжимается независимо от других. Разумеется, кластеры не должны быть слишком большие, чтобы не занимать слишком много памяти при попытке что-либо прочитать или записать по произвольному смещению: максимальный размер логического кластера, поддерживаемый reiser4 – 64K. Это обстоятельство несколько снижает степень сжатия данных из-за невозможности создать обширный словарь в процессе компрессии.

Логический кластер индекса I – это множество байт данного файла, смещения которых лежат в отрезке [I*S, (I+1)*S-1], где S – размер логического кластера. В настоящее время Reser4 поддерживает кластеры с размерами 4K, 8K, 16K, 32K и 64K, но не меньше PAGE_SIZE). Логический кластер называется частичным, если в нем задействовано меньше S байт.

Страничный кластер индекса I – это последовательность страниц, содержащих сырые (несжатые и незашифрованные) данные соответствующего логического кластера. Страничный кластер присутствует в памяти во время чтения или записи файла.

Дисковый кластер индекса I – это последовательный набор итемов какого-либо типа, первый из которых имеет ключ со смещением, которое вычисляется как функция от I (этим заведует специальный метод итем-плагина). Размер дискового кластера определяется как S*N, где N - коэффициент растяжения крипто-алгоритма, которым зашифрован данный файл (N = 1 для всех симметричных алгоритмов).

В настоящее время “кластеризованными” являются только ctail-итемы, структура которых определена в plugin/item/ctail.h:

```
typedef struct ctail_item_format {
      d8 cluster_shift;      /* Двоичный логарифм размера дискового
                              * кластера */
      d8 body[0];            /* Тело итема */
} __attribute__ ((packed)) ctail_item_format;
```

Каждый дисковый кластер представляет собой разбитые на ctail-итемы сжатые и зашифрованные данные логического кластера в определенном формате, который не представлен какой-либо структурой данных и имеет следующий вид:

```
      данные {дополнение контрольный_байт контрольная_сумма}
```

Дополнение служит для выравнивания сжатых данных перед шифрованием, чтобы итоговый размер был кратен размеру блока крипто-алгоритма. В контрольном байте хранится размер дополнения, увеличенный на 1 (фактически это размер участка, который потребуется отсечь перед декомпрессией). Контрольная сумма – это

adler32 от выровненных и зашифрованных данных.

Контрольная сумма страхует драйвер от попытки применить декомпрессию к некорректным данным (последний случай чреват фатальными последствиями, т. к. допускается использование небезопасных алгоритмов сжатия (которые, как правило, являются наиболее быстрыми). Контрольная сумма добавляется только в том случае, когда компрессия была принята (в этом случае итоговый размер данных с учетом контрольной суммы должен быть строго меньше чем размер дискового кластера). Если же данные логического кластера плохо сжимаемы, то результат компрессии отвергается, и к выровненному и зашифрованному логическому кластеру контрольная сумма не добавляется. При таком подходе смежные дисковые кластеры не “накладываются” друг на друга по смещениям в их ключах. Другое важное преимущество состоит в том, что по каждому дисковому кластеру можно сразу определить, была ли произведена компрессия, или нет.

При чтении файла по какому-либо смещению в памяти размещаются страничные кластеры соответствующих индексов. При запросе на чтение какой-либо страницы файловая система размещает в памяти целиком весь страничный кластер (по смещению конструируется ключ, последовательно находятся и компануются все ctail-итемы соответствующего дискового кластера, после чего происходит расшифровка, декомпрессия и заполнение страниц сырыми данными). При этом попытка прочитать немного больше положенного (менеджер виртуальной памяти ничего не знает о кластерах) не наносит ущерба и вписывается в общую концепцию упреждающего чтения.

Подробнее об этом плагине можно почитать в [4].

Compound directory item

Составной итем каталога состоит (в отличие от single directory item, который в настоящее время не используется и описан здесь не будет) из нескольких элементов каталога. Был введен с целью повышение эффективности использования дискового пространства. Дело в том, что все элементы одной директории имеют в своих ключах одинаковый фрагмент – ObjectID родительского каталога. Компоновка составного итема из нескольких элементов одной директории позволяет сохранять указанный фрагмент ключа только единожды и, тем самым, сэкономить дисковое пространство. Это решение является особой формой сжатия ключей, т.к. их полноценное сжатие в

версии 4.0 не реализовано. Заметьте также, что на диске ключи хранятся не выровненными, что, по крайней мере на некоторых архитектурах, повышает нагрузку на CPU при их обработке, но, опять же, экономит место.

Дисковая структура CDE-итема такова:

Item header	Entry headers		Entry bodies	
count of entries	hash	offset	key	name

Форматы его компонентов определены в plugin/item/cde.h:

```
typedef struct cde_unit_header {
      de_id hash;                   /* Часть ключа (2 последних элемента) */
      d16 offset;                   /* Смещения тела элемента каталога */
} cde_unit_header;

typedef struct cde_item_format {
      d16 num_of_entries;           /* Количество элементов каталоге */
      cde_unit_header entry[0];     /* Массив заголовков элементов */
} cde_item_format;

/* Формат элемента каталога (directory entry) */
typedef struct directory_entry_format {

      /* Ключ stat-data итема описываемого объекта. Нет нужды хранить
       * его целиком, т.к. это всегда ключ stat-data и, следовательно,
       * тип  и offset могут быть опущены. Однако из-за возможности
       * применения других схем назначения ключей здесь
       * зарезервировано место для целого ключа */
      obj_key_id id;

      /* Имя объекта - строка сNULL- байтом в конце*/
      d8 name[0];
} directory_entry_format;
```

Выводы

Обобщая изложенный выше материал, можно сказать, что регулярный файл/каталог в reiser4 состоит из объектов трех типов:

- элемент каталога; содержит имя объекта и ключ его stat-data итема
- stat-data итем; атрибуты
- один или несколько итемов, содержащих тело объекта: для регулярного

файла это extent/tail/ctail итемы, для каталога – CDE-итемы; специальные файлы (устройства, FIFO, и т.д.) тела не имеют.

Все итемы, принадлежащие одному файлу, имеют в своих ключах одинаковые dirid и objectid.

Журналирование

Reiser4 поддерживает режим полного журналирования данных и метаданных, обеспечивая также некоторые расширенные возможности.

Дело в том, что большинство файловых систем осуществляют кэширование записи – модифицированные данные не сбрасываются немедленно на диск, а накапливаются в кэше. Это позволяет ФС не только лучше контролировать дисковое планирование, но и формировать длинные I/O запросы, которые современными жесткими дисками обрабатываются существенно быстрее, нежели группа коротких. В случае сбоя системы из-за этого будут не просто потеряны недавние изменения – механизм кэширования может поменять местами запросы на запись, в результате чего более новые данные окажутся записанными, а старые – потеряны. Это может стать серьезной проблемой для приложений, делающих несколько зависимых модификаций, часть из которых будет утеряна, а часть – нет. Такие приложения требуют от ФС гарантировать, что либо все либо ни одно изменение не переживет сбой.

Зависимые модификации также могут возникать когда приложение читает модифицированные данные и затем производит вывод, например:

1. Процесс 1 пишет в файл A
2. Процесс 2 читает из файла A
3. Процесс 2 пишет в файл B

Очевидно, что файл B может зависеть от файла A, и, если стратегия кэширования поменяет порядок фиксации изменений в этих файлах, после сбоя приложения могут оказаться в некорректном состоянии.

Атом – набор блоков, модификации которых должны быть атомарно записаны на диск. Каждое незафиксированное на диске изменение объекта ФС есть атом.

Например, если приложение добавляет данные в конец файла, в атом будет включен блок, содержащий сами новые данные, блок со stat-data итемом, содержащий длину файла и блок, хранящий tail/ctail/extent итем. Если размещен новый блок – в атом добавится также суперблок, хранящий счетчик свободного места, и блок битовой карты (в действительности алгоритм журналирования суперблока и битовых карт несколько более сложен, см. ниже).

Транскрэш – набор операций, все или ни одна из которых не переживут сбой системы.

Существует 2 типа транскрэшей: read-write и write-only. Если приложение пишет в модифицированный, но еще не зафиксированный на диске блок, то атомы, воплощающие две эти модификации, объединяются в write-only транскрэш. Если приложение читает данные незафиксированного блока, а затем осуществляет запись – эти 2 атома сливаются в read-write транскрэш.

Более подробно о реализации описанных выше механизмов можно почитать в комментариях файлов txnmgr.[ch], а также в [3].

Низкоуровневые механизмы журналирования реализованы в wander.[ch] и таже достаточно необычны. Начать хотябы с того, что reiser4 не имеет журнала (выделенной области на диске) в обычном его понимании. “Странствующие” журнальные блоки размешаются произвольно, в любом месте файловой системы. А вместо того, что бы записывать журналируемый блок дважды (один раз – в “странствующее” местоположение, второй – в реальное), reiser4 может записать блок в новое место, а затем обновить указатель в его родителе. Казалось бы, какая разница – ведь модификацию родителя все равно придется включать в транзакцию, однако при журналировании хотя бы трех блоков с общим родителем выгода становится очевидной. Традиционное решение называется перезаписью блоков, а описанное – переразмещением.

Решение о переразмещении или перезаписи принимается из соображений улучшения производительности. Записывая журналируемые блоки в новые места, ФС избегает необходимости вносить копию каждого блока в журнал. Однако в случае, когда начальная позиция переразмещаемого блока является оптимальной, смена его координат может увеличить фрагментацию.

В дальнейшем под *фиксацией* (commit) атома будем понимать запись составляющих его блоков в странствующие положения, а под *сбросом* (flush) – запись блоков по реальным координатам. Понятно, что для переразмещаемого набора атома эти две стадии совпадают.

Перезаписываемый набор атома содержит все грязные блоки, не принадлежащие переразмещаемой группе (т.е. блоки, не имеющие грязного родителя, для которых перезапись будет лучшим выходом). “Странствующая” копия каждого блока записывается как часть журнала до сброса атома, и заменяет оригинальное содержимое блока после сброса. Заметьте, что суперблок считается родителем корневого узла, а bitmap-блоки не имеют родителей. Следовательно, они всегда будут обрабатываться как часть перезаписываемого набора (можно также определить *минимальный перезаписываемый набор*, который аналогичен обычному за исключением следующих условий: если по крайней мере 3 грязных блока имеют общего чистого родителя, тогда их родитель добавляется в этот минимальный набор а сами блоки перемещаются в relocate-группу. Эта оптимизация будет сохранена для последующих версий).

В зависимости от системы и рабочей нагрузки можно выбрать одну из трех политик журналирования:

- Always Relocate: эта политика включает блок в переразмещаемый набор в любом случае, сокращая количество блоков, сбрасываемых на диск (оптимизирует запись).
- Never Relocate: эта политика отключает переразмещение. Блоки всегда пишутся в оригинальное местоположение с журналированием через перезапись, дерево не изменяется (оптимизирует чтение).
- Левый сосед: эта политика перемещает блок в ближайшее к его левому соседу возможное место (в порядке дерева). Если данная позиция занята некоторым блоком выполняющегося атома, политика делает блок членом перезаписываемого набора.

Журналирование метаданных

Журналирование метаданных – это ограниченный режим работы механизма журналирования, при котором атомарной записью защищаются только метаданные файловой системы. В этом режиме блоки данных файла (неформатированные узлы)

не затрагиваются механизмом журналирования и, следовательно, не нуждаются в записи на диск как результате фиксации транзакции. В этом случае блоки данных файла не рассматриваются членами relocate или overwrite наборов, т.к. они не участвуют в протоколе атомарных обновлений, а единственными причинами их сброса на диск являются переполнение памяти и возраст.

Обработка bitmap-блоков

Reiser4 размещает временные блоки для "странствующего" журналирования. Это значит, что существуют различия между содержимым фиксированного bitmap-блока, которое должно быть восстановлено после сбоя, и содержимым рабочего bitmap-блока, который используется для поиска/выделения свободных блоков.

Для каждого bitmap-блока в памяти хранятся 2 версии: WORKING_BITMAP и COMMIT_BITMAP.

Рабочий bitmap используется просто для поиска свободных блоков, если некоторый бит сброшен в рабочей карте, то соответствующий блок может быть выделен. Рабочая карта обновляется при каждом выделении блока. Commit-карта отражает изменения, выполненные в уже совершенных атомах либо в атоме, который выполняется в данный момент. Commit-карта обновляется только при фиксации атома, т.е. состояние блоков, выделенных журналу на время обработки атома, в ней никогда не отражается.

Наличие двух битовых карт в памяти очень выгодно, т.к. это позволяет нескольким атомам модифицировать один bitmap-блок.

Другой дефицитный ресурс в reiser4 – суперблок, содержащий счетчик свободных блоков. К нему применяется аналогичная техника, позволяющая многим атомам модифицировать этот счетчик (см. ниже).

Странствующее журналирование

Алгоритм журналирования reiser4 размещает и записывает странствующие блоки и поддерживает дополнительные дисковые структуры атома – wander-записи (каждая занимает 1 блок), содержащие общую информацию о транзакции и таблицу отображений странствующих блоков на их реальные координаты.

Дисковая структура транзакции такова:

Transaction header,	Wander records [0,n]	
1-st block	Wander record header	wander entries [0,n]
tx_header	wander_record_header	wander_entry

См. wander.h:

```
struct tx_header {

	/* Магическая строка делает первый блок в транзакции отличным
	 * от других журналируемых блоков, это должно помочь fsck */
	char magic[TX_HEADER_MAGIC_SIZE];

	/* ID транзакции */
	d64 id;

	/* Общее количество wander-записей (включая эту, tx head) */
	d32 total;

	/* Выравнивает предыдущее поле по 8-байтной границе, всегда 0 */
	d32 padding;

	/* Указатель на заголовок предыдущей транзакции */
	d64 prev_tx;

	/* Положение следующей wander-записи */
	d64 next_block;

	/* Зафиксированная версия счетчика свободных блоков */
	d64 free_blocks;

	/* Количество файлов и следующий свободный ObjectID
	 * журналируются отдельно от суперблока */
	d64 nr_files;
	d64 next_oid;
};

struct wander_record_header {

	/* Если не известно положение wander-записей, эта строка поможет
	 * fsck найти их. */
	char magic[WANDER_RECORD_MAGIC_SIZE];

	/* ID транзакции */
	d64 id;

	/* Общее количество wander-записей в транзакции */
	d32 total;
```

```
     /* Количество блоков в транзакции */
     d32 serial;

     /* number of next block in commit */
     d64 next_block;
};

/* Остаток блока, содержащего wander запись, заполняется этими
 * элементами, а неиспользованное место - нуями*/
struct wander_entry {
     d64 original;          /* Оригинальное положение блока */
     d64 wandered;          /* Странствующее положение блока */
};
```

Для управления журналом в reiser4 существуют 2 блока, имеющие фиксированное положение на диске: заголовок журнала и journal footer. Атомарная запись заголовка журнала показывает, что транзакция зафиксирована (т.е. переживет сбой), а запись journal footer'a – что выполнены все постфиксационные записи (т.е. транзакция полностью завершена, все блоки записаны по своим местам). После удачной записи footer'a все странствующие блоки и wander-записи освобождаются. См. wander.h:

```
/* Формат блока заголовка журнала */
struct journal_header {
     /* Положение заголовка последней зафиксированной транзакции */
     d64 last_committed_tx;
};

/* Формат journal footer блока */
struct journal_footer {

     /* Положение последней сброшенной транзакции. Этот указатель
      * не является истинным после того, как транзакция, на которую
      * он указывает, сброшена, а используется только в процессе
      * восстановления для определения конца дискового списка
      * зафиксированных транзакций, которые не были успешно
      * сброшены. */
     d64 last_flushed_tx;

     /* Счетчик свободных блоков во время сброса транзакции
      * записывается в journal footer, а не в суперблок, т.к. он
      * журналируется отлично от других полей суперблока (например,
      * указателя на корень дерева) */
     d64 free_blocks;

     /* Количество файлов и максимальный OID также журналируются
      * отдельно от суперблока */
     d64 nr_files;
```

```
        d64 next_oid;
};
```

Процесс фиксации атома включает в себя несколько стадий:

1. Подсчитывается размер перезаписываемого набора атома.
2. Вычисляется необходимое количество wander-записей, размещаются необходимые для них блоки.
3. Размещаются странствующие блоки и заполняются wander-записи.
4. Странствующие блоки и wander-записи направляются на запись.
5. Ожидается завершение I/O.
6. Обновляется заголовок журнала: указатель last_commited_tx устанавливается на блок tx_header действующей транзакции, модифицированный блок заголовка журнала направляется на запись и ожидается завершение I/O.

Сброс атома:

1. Перезаписываемый набор атома пишется по оригинальным координатам.
2. Ожидается завершение I/O.
3. Обновляется footer журнала: указатель last_fulshed_tx устанавливается на блок tx_header текущего атома. Footer-блок направляется на запись.
4. Ожидается завершение I/O.
5. Освобождается дисковое пространство, выделенное странствующим блокам и wander-записям (Вносятся изменения в рабочие битовые карты, на диск ничего не пишется).

Нетрудно понять, что когда процедура восстановления ищет незавершенные транзакции, она сравнивает значения полей last_commited_tx (заголовок журнала) и last_flushed_tx (journal footer), и если они не равны – начинает движение по кольцевому списку wander-записей, сбрасывая все, что ФС успела зафиксировать до сбоя.

Более подробно об описанных выше механизмах можно почитать в комментариях файлов wander.[ch] и в [3].

Плагинная инфраструктура

Все плагины reiser4 классифицируются по нескольким типам. Плагины одного

типа называются его *инстанциями* (возможно, более правильно сказать *экземплярами*). Метка в виде пары (type_label, plugin_label) уникальна и является глобально устойчивым, видимым пользователю идентификатором плагина. Внутри ядра поддерживаются массивы, индексами в которых и являются эти числа. Также поддерживаются статические словари, являющиеся отображением меток плагинов на внутренние идентификаторы типа reiser4_plugin_type, хранящиеся в объектах ФС.

Метки плагинов имеют значение для пользовательского интерфейса, назначающего плагины объектам, и в будущем станут использоваться для их динамической загрузки. Идентификатор типа reiser4_plugin_type является индексом во внутреннем статическом массиве plugins[].

Объект ФС, в соответствие которому поставлен плагин некоторого типа, назван, без лишних премудростей, субъектом этого типа и его конкретной инстанции. С каждым субъектом плагин может хранить некоторое состояние (для этого в stat-data есть специальное расширение reiser4_plugin_stat). Например, состояние директорного плагина (является инстанцией объектного типа) есть указатель на хэш-плагин.

В дополнение к числовому идентификатору каждый тип и каждая инстанция имеют текстовую метку (короткую строку) и определение (длинную строку), жестко кодированные в массиве plugins[]. По этой паре плагин также может быть надежно идентифицирован.

С каждым inode (открытым файлом) ассоциирован целый набор плагинов. Хранить указатели на них в каждом inode – непростительная потеря памяти. Вместо это reiser4 поддерживает несколько глобальных структур данных типа struct plugin_set, каждая из которых хранит набор плагинов для объекта конкретного типа. Inode содержит только указатель на plugin_set своего типа.

Структуры данных, используемые плагинной инфраструктурой, а также возможные типы и реализованные инстанции описаны в файлах:

- plugin/plugin.[ch]
- plugin/plugin_header.h
- plugin/plugin_set.[ch].

Заключение

В vanilla-ядра драйвер reiser4 пока не включается, постоянно доступен только в -mm ветке Эндрю Мортона (Andrew Morton): www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6

На ftp-сайте разработчиков (ftp://namesys.com/pub/reiser4-for-2.6) также можно найти неофициальные reiser4-патчи для некоторых vanilla-ядер, правда выходят они с некоторым опозданием и практически не сопровождаются.

Источники

1. Исходные тексты драйвера reiser4 для linux-2.6.23
2. Hans Reiser “Reiser4 whitepaper”: www.namesys.com/v4.html
3. Joshua MacDonald, Hans Reiser, Alex Zarochentcev, "Reiser4 transaction design document": www.namesys.com/txn-doc.html
4. Edward Shishkin, “Reiser4 cryptcompress regular files”: www.namesys.com/cryptcompress_design.html

Свежую версию этого документа, а также аналогичные по тематике статьи и переводы можно найти на www.filesystems.nm.ru