Дисковая структура ReiserFS

Пешеходов А. П. aka fresco, Понедельник, 31 Октябрь 2005, 12:48

Примечания

Статья составлена по документу Florian Buchholz “ The structure of the Reiser file system”, по комментариям в исходных текстах ReiserFS, а также по консультациям с Владимиром Савельевым.

При написании статьи использованы исходники драйвера ReiserFS ядра Linux-2.6.13.3.

Введение

Reiser File System была создана Гансом Рейзером с целью увеличения производительноти (по сравнению с ext2), создания эффективной схемы управления дисковым пространством и обеспечения улучшенной (относительно существующих файлоывх систем) обработки больших каталогов. ReiserFS использует сбалансированны деревья для хранения файлов и директорий, а также обеспечивает журналирование.

Этот документ описывает дисковую структуру ReiserFS версии 3.6. Здесь не рассматриваются алгоритмы балансировки filesystem tree, выполнения журналирования, операций управления файлами, каталогами и пр.

Блоки

Раздел ReiserFS представляет собой набор блоков фиксированного размера. Блоки нумеруются последовательно начиная с нулевого. Максимально доступное количество блоков на одном разделе – 2^32.

Раздел начинается с 64-х килобайт неиспользуемогопространства, оставленного под загрузчики, дисковые метки и прочие служебные надобности. Далее следует суперблок. Суперблок содержит важную информацию о разделе, например размер блока и местоположение корневого узла и journal node. Номер блока, содержащего суперблок, варьируется в зависимости от размера блока файловой системы, однако он всегда начинается с 65536-го байта раздела. Размер блока reiserfs в Linux по умолчанию равен 4 kb. Таким образом, суперблок содержится в 16-м блоке. Суперблок – один на весь раздел.

Сразу за суперблоком следует блок, содержащий битовую карту свободнго места. Количество блоков, отслеживаемых картой, напрямую зависит от размера блока. Большой раздел может иметь несколько bitmap-блоков.

За первым bitmap-блоком должен быть журнал, однако суперблок содержит более точную (для нестандартных случаев) информацию о его местоположении.

Суперблок

См. include/linux/reiserfs_fs.h

```
/* Параметры журнала */
struct journal_params {
 __le32 jp_journal_1st_block; /* Первый блок журнала на этом разделе */
 __le32 jp_journal_dev; /* Номер устройства, содрежащего журнал */
 __le32 jp_journal_size; /* Размер журнала */
 __le32 jp_journal_trans_max; /* Максимальное количество блоков в транзакции */
 __le32 jp_journal_magic; /* Случайный magic-номер (присваивается при создании ФС */
 __le32 jp_journal_max_batch; /* Максимальное количество блоков в транзакции */
 __le32 jp_journal_max_commit_age; /* Максимальный возраст (в секундах) асинхронного
       пакета (commit) */
 __le32 jp_journal_max_trans_age; /* Максимальный возраст (в секундах) транзакции */
};

/* Формат суперблока для reiserfs v 3.5.x, где x >=10 */
struct reiserfs_super_block_v1 {
 __le32 s_block_count; /* Количество блоков */
 __le32 s_free_blocks; /* Количество свободных блоков */
 __le32 s_root_block; /* Номер блока, содержащего корневой узел дерева файловой
       системы */
 struct journal_params s_journal; /* Параметры журнала (см. выше) */
 __le16 s_blocksize; /* Размер блока (в байтах) */
 __le16 s_oid_maxsize; /* Максимальный размер массива Object ID (подробнее см.
       комментарии к reiserfs_get_unused_objectid() в fs/reiserfs/objectid.c */
 __le16 s_oid_cursize; /* Текущий размер массива Object ID */
 __le16 s_umount_state; /* Unmount статус
 1 – если ФС была размонтирована
 2 – если нет */
 char s_magic[10]; /* Reiserfs magic string
 "ReIsErFs", "ReIsEr2Fs" или "ReIsEr3Fs" */
 __le16 s_fs_state; /* Это поле используется fsck для отметок о том, какая фаза
       восстановления завершена */
 __le32 s_hash_function_code; /* Определяет, какая хэш-функция будет использоваться
       для сортировки имен в каталогах*/
 __le16 s_tree_height; /* Высота дерева */
 __le16 s_bmap_nr; /* Количество bitmap-блоков, необходимых для отслеживания
       каждого блока в файловой системе */
 __le16 s_version; /* Версия reiserfs (актуальна только на ФС с нестандартым
       журналом) */
 __le16 s_reserved_for_journal; /* Размер (в блоках) журнала на основном
       устройстве, который мы должны сохранить после создания ФС с нестандартым
       журналом */
};

/* Дисковый суперблок */
struct reiserfs_super_block {
```

```
struct reiserfs_super_block_v1 s_v1; /* См. выше */

__le32 s_inode_generation; /* Поколение inode – счетчик, увеличивается при
    каждом удалении */
__le32 s_flags; /* В настоящее время используется только подсистемой
    inode-attributes, если она разрешена */
unsigned char s_uuid[16]; /* Уникальный идентификатор ФС */
unsigned char s_label[16]; /* Метка тома (volume lable) */
char s_unused[88]; /* используется mkreiserfs и reiserfs_convert_objectid_map_v1() */
};
```

Поле s_inode_generation используется для работы reiserfs на NFS-сервере. NFS-клиент получает от сервера handle, содержащий ID файла и generation counter. Допустим, что далее файл на сервере удаляется, а через некоторое время создается новый, получающий тот же ID (reiserfs, как большинство других файловых систем, присваивает новым файлам освободившиеся идентификаторы удаленных). Клиент, не зная этого, делает новый запрос с handle удаленного файла. Если бы не s_inode_generation, сервер не смог бы понять, что handle, переданный клиентом, относится к несуществующему уже файлу, т.к. он ищет inode файла в кэше inodes по ID, и он нашел бы новый файл. Однако s_inodes_generation при новом файле отличается от того, что был при старом, то сервер может понять, что ему передан устаревший NFS-handle.

Для синхронизации процессов, ожидающих балансировки дерева, reiserfs имеет не хранимый на диске счетчик fs_generation, используемый для того, что бы определить, изменилась файловая система с момента последнего чтения счетчика. Балансировка – сложный и длительный процесс, подготовка к которому часто требует чтение с диска нескольких блоков. Т.к. reiserfs не имеет locking-механизма, ни что не мешает другому процессу изменить дерево, пока наш процесс ждет завершения ввода-вывода. В таком случае может оказаться, что все данные, подготовленные первым процессом, уже не актуальны, и надо сделать все заново. fs_generation увеличевается при каждом изменении в дереве. Процесс, начинающий подготовку к балансировке, запоминает fs_generation, а перед началом самой балансировки сравнивает этой значение с новым. Если fs_generation изменился – подготовка выполняется заново.
[newpage]

Bitmap-блоки

Bitmap-блоки – это простые битовые карты, где каждый бит сопоставляется с номером блока. Один такой блок может адресовать 8*blocksize блоков. Если бит установлен – блок занят, если сброшен – свободен.

File System Tree

Файловая система reiserfs предствлена в виде сбалансированноо дерева внешнего поиска (B+, или S+ дерево, как оно называется в документации на reiserfs). Дерево состоит из внутренних и листовых узлов. Каждый узел (node) – это дисковый блок. Каждому объекту (называемому item (запись)) в reiserfs (файлу, каталогу, stat-item)назначается уникальный

ключ, аналогичный номеру inode в других файловых системах. Внутренние узлы, главным образом, состоят из ключей и указателей на узлы-потомки. Указателей всегда на один больше, чем ключей. Если P[i] указывает на объект, имеющий ключ меньше K[i], то P[i] – на объект с ключем >= K[i].

Заголовки блоков

Каждый дисковый блок, содержащий внутренний или листовой узел, начинается с заголовка блока.

См. include/linux/reiserfs_fs.h

```
struct block_head {
 __le16 blk_level; /* Уровень блока в дереве */
 __le16 blk_nr_item; /* Количество ключей/записей в блоке */
 __le16 blk_free_space; /* Свободно (в байтах) */
 __le16 blk_reserved; /* Не используется */
 struct reiserfs_key blk_right_delim_key; /* Раньше использовался для листьев,
       сохранен для совместимости */
};
```

Ключи

Ключи используются в reiserfs в качестве уникальных идентификаторовзаписей, а также для определения положения записей в дереве. Ключ состоит из четырех объектов: ID родительского каталога, ID объекта (object ID), смещения объекта и его типа. Примечательно, что фактически идентификатор объекта – это только одна часть ключа. Directory ID нужен, для группировки объектов, принадлежащих одному каталогу, и размещения большей их части в одном поддереве. Хранить смещение необходимо потому, что косвенная запись (см. ниже) может содержать самое большее (blocksize-48)/4 указателей на неформатированне блоки. Для размера блока в 4 kb это будет означать, что размер файла ограничен 4048 kb. Что бы обеспечить возможность обработки больших файлов, для их описания используется множество ключей. Все поля таких ключей одинаковы, за исключением offset, которое указывает на положение в файле той его части, на которую ссылается данный ключ.

В reiserfs до версии 3.5 поля type и offset были 32-хбитными величинами. Из-за этого размер файла был ограничен приблизительно четырьмя Gb (точнее 2^32 плюс данные еще одной косвенной записи плюс “хвост” (tail)). Для снятия этого ограничения в версии 3.6 поле offset было увеличено до 60 бит, а поле type – сокращено до 4 бит. Теперь теоретически доступны файлы размером до 2^60 байт, однако фактически мы можем адресовать только 2^32 блоков с максимально возможными 2^16 байт на блок – то есть не более 2^48 байт на файл. Для совместимости со старыми версиями reiserfs (и старыми ключами) был введен достаточно запутанный интерфейс – т.к. сам ключ не несет в себе номера версии. Для решения этой проблемы ранее зарезервирванны последние 16 бит заголовка записи (item header, см. ниже) теперь являются индикатором номера версии. Для ключей в листьях определить версию доволбно просто, однако если нужно получить версию ключа во внутреннем узле, то придется

спуститься вниз по дереву – к соответствующему листу.

Идентификатор типа
Тип v1 v2
stat-item 0 0
прямая запись 0xfffffffe 1
косвенная запись 0xffffffff 2
директория 500 3
any 555 15

Отсюда следует, что stat-item будет всегда определяться как запсиь с KEY_FORMAY_1 – она имеет идентификатор типа = 0 в обоих версиях ключа.

См. include/linux/reiserfs_fs.h

```
struct in_core_key {
 __u32 k_dir_id; /* Object ID родительского каталога */
 __u32 k_objectid; /* Object ID */
 __u64 k_offset; /* Смещение (в байтах) части объекта, на которую ссылается ключ */
 __u8 k_type; /* Тип записи. */
};

struct cpu_key {
 struct in_core_key on_disk_key; /* Дисковый ключ */
 int version; /* версия */
 int key_length; /* = 3 во всех случаях, используется для direct->indirect и
        indirect->direct преобразования*/
};
```

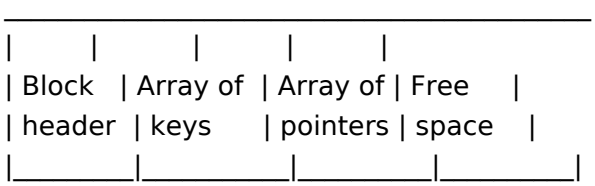
Только stat-item имеет поле offset=0. Файлы (прямые и косвенные записи) и каталоги всегда начинаются со смещения 1 для того, что бы в результате сортировки они расположились в листе позади stat-записей. Для директорных записей поле offset хранит хэш имени и номер поколения крайнего левого directory header (см. ниже) записи каталога.

При сравнении ключей их поля стравниваются в таком порядке: сперва directory ID, если равны – object ID, offset, type. Код reiserfs в Linux генерирует предупреждение (warning), если дело доходит до сравнения типов, т.к. это сравнение не имеет смысла со струтурной точки зрения. Единственная ситуация, когда типы ключей должны быть протестированы – tail conversion, когда прямые записи становятся косвенными или наоборот.

Внутренние узлы

Блок внутреннего узла состоит из заголовка блока, ключей и указателей на узлы-потомки. В начале узла располагается заголовок блока, затем все ключи, отсортированныепо значению, далее – указатели на узлы-потомки.

См. include/linux/reiser_fs.h



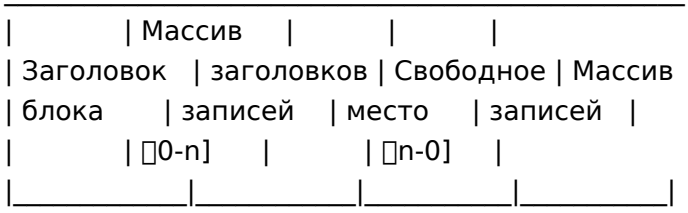
Поле blk_level заголовка блока для внутренних узлов всегда больше 1. blk_nr_item в заголовке блока означает количество ключей в блоке (а не количество ключей и указателей). Указателей всегда на 1 больше, чем ключей!

См. include/linux/reiserfs_fs.h

```
/* Указатель на блок-потомок */
struct disk_child {
 __le32 dc_block_number; /* Номер блока, содержащего потомка */
 __le16 dc_size; /* Использовано байт в блоке */
 __le16 dc_reserved; /* Зарезерсиврован */
};
```

Пусть имеем ключ n (его смещение в блоке равно 24 + n*16 байт) и всего k ключей в блоке. Тогда левый указатель, относящийся к этому ключу (и определяющий его меньшего потомка) может быть найден по смещению 24 + k*16 + n*8 байт.
[newpage]
Листья

Листья находятся на самом нижнем уровне S+ дерева. Все данные содержаться внутри самого узла (исключая данные, адресуемые косвенной записью и располагающиеся в неформатированнх узлах (или блоках, unformated nodes). Листья представлены заголовком блока, заголовками записей и собственно записями.



Примечательно, что свободное пространство в блоке расположено между последним

заголовком записи и первой записью (т.е. записи идут в обратном порядке). Таким образом, новый заголовок записи и сама запись могут быть довольно просто добавлены в лист без его перегруппировки Заметьте также, что записи имеют переменную длину.

Заголовки записей

Заголовок записи содержит ключ, смещение записи в листе и ее размер.

См. include/linux/reiser_fs.h

```
struct item_head {
 struct reiserfs_key ih_key;
 union {
 __le16 ih_free_space_reserved; /* Количество свободных байт в последнем
       неформатированнм узле для косвенной записи , 0xffff для прямой и
       stat-записи */
 __le16 ih_entry_count; /* Количество элементов каталога для директорной записи */
 } u;
 __le16 ih_item_len; /* Размер записи */
 __le16 ih_item_location; /* Смещение начала записи в блоке */
 __le16 ih_version; /* 0 для всех старых записей, 2 для новых. Старший бит
       устанавливается fsck на время выполнения */
};
```

В комментариях к определению этой структуры сказано, что поле ih_version равно 2 для всех новых записей. Однако константа KEY_FORMAT_3_6, используемая для инициализации поля, определена как 1.

Записи

Запси (items) наконец содержат собственно данные. Существует 4 типа записей: stat-записи, записи каталога, прямые и косвенные записи. Файл может быть представлен одной или несколькими прямыми или косвенными записями – в зависимости от его размера. Каждому файлу или директории предшествует stat-запись.

Stat-items

Stat-запись содержит метаданные для файлов и директорий. Ключи, принадлежащие stat-записи, имеют нулевые поля offset и type что бы всегда располагаться впереди других записей, принадлежащих тому же самому “номеру inode”. По тем же причинам, по которым в reiserfs существуют 2 версии ключей, поддерживаются также и 2 версии stat item. В новой версии поле size увеличено с 32-х до 64-х бит, а также, по некоторым причинам, были увеличены размеры полей, отвечающих за хранение количества жестких ссылок, user ID и group ID – с 16-ти до 32-х бит. Размер самого stat item увеличился с 32-х байт в первой версии до 44-х во второй.

См. include/linux/reiserfs_fs.h

```
/* Старая версия stat data размером 32 байта */
struct stat_data_v1 {
 __le16 sd_mode; /* Тип и права доступа к файлу */
 __le16 sd_nlink; /* Количество жестких ссылок */
 __le16 sd_uid; /* Владелец (owner) */
 __le16 sd_gid; /* Группа (group) */
 __le32 sd_size; /* Размер файла */
 __le32 sd_atime; /* Времы последнего доступа */
 __le32 sd_mtime; /* Время последней модификации */
 __le32 sd_ctime; /* Время последней модификации stat data */
 union {
 __le32 sd_rdev; /* Номер устройства */
 __le32 sd_blocks; /* Количество выделенных файлу блоков */
 } u;
 __le32 sd_first_direct_byte; /* Смещение первого байта файла, хранимого в прямой
       записи: если 1 – это символическая ссылка, если (__u32)0 – это не
       прямая запись. Существование этого поля меня реально достало
       (grates). Заменим его вместе с макросами, основанными на sd_size и
       нашей политикой запрещение хвостов (tail supression policy,
       очевидно имеется в виду механизм, стоящий за опцией notail – пер.)
       Когда-нибудь. Ганс.*/
};

/* Дисковый stat item (наша версия дискового inode UFS за вычетом адресных блоков */
struct stat_data {
 __le16 sd_mode; /* Тип и права доступа к файлу */
 __le16 sd_attrs; /* Inode флаги */
 __le32 sd_nlink; /* Количество жестких ссылок */
 __le64 sd_size; /* Размер файла */
 __le32 sd_uid; /* owner */
 __le32 sd_gid; /* group */
 __le32 sd_atime; /* ... */
 __le32 sd_mtime; /* ... */
 __le32 sd_ctime; /* ... */
 __le32 sd_blocks; /* количество блоков, выделенных файлу */
 union {
 __le32 sd_rdev; /* Номер устройства */
 __le32 sd_generation; /* Поколение файла */
 } u;
};
```

Поле sd_mode – битовая маска в стиле struct stat. Только регулярные файлы и каталогии имеют другие записи, ассоциированныесо stat item. Во всех других случаях (сокет,

символическая ссылка, блочное и символьное устройство, а также FIFO) stat item представляет собой все содержимое файла.

Поле sd_rdev применяется для специальных файлов. Оно хранит номер устройства (или сокета). sd_generation примеятся во всех других случаях и означает поколение inode для файла/каталога/ссылки. Поле sd_first_direct_byte, как и хотел Hans Reiser, во второй версии более не используется.

Directory items

Запись каталога описывает директорию. Если каталог имеет слишком много элементов и не умещается в одной записи, он разбивается на несколько directory items. Directory item состоит из 2-х массивов: заголовков элементов каталога и имен файлов, растущих к середине.

См. include/linux/reiserfs_fs.h

```
struct reiserfs_de_head {
 __le32 deh_offset; /* Третий компонент ключа элемента каталога – значение хэша и
      номер поколения */
 __le32 deh_dir_id; /* Object ID родительского каталога */
 __le32 deh_objectid; /* Object ID данного элемента */
 __le16 deh_location; /* Смещение имени файла в данной записи */
 __le16 deh_state; /* Бит 1 указывает, что этот элемент содержит stat- data
      (не используется; бит 2 – элемент доступен (1) или скрыт (0) */
};
```

Имя файла – простая ASCII строка, заканчивающаяся нулем. Поле offset просто неверно названо – оно содержит значение хэша для имени файла. Биты [7-30] – это хэш, [0-6] – поколение для файлов с одинаковым хэшем, бит 31 не используется. Хэш используется в reiserfs для поиска имен файлов и каталогов, отсортированныхпо полю offset. В настоящий момент доступны 3 хэш-функции: keyed tea хэш, rupasov-хэш и r5-хэш. Задача хэш-функции – генерировать разные значения для разных строк с как можно меньшим количеством коллизий. В Linux-реализации reiserfs по умолчанию используется r5-хэш.
[newpage]
Direct items

Прямые записи содержат либо весь маленький файл, либо “хвост” большого файла. Для маленьких файлов вся другая информация может быть найдена (через заголовок записи) в stat-item для данного файла.

Indirect items

Косвенные записи содержат указатели на неформатированне блоки, принадлежащие файлу. Каждый указатель – 4-хбайтное число, содержащее номер блока. Косвенная запись, занимающая весь лист дерева, может содержать (blocksize-48)/4 указателей (48 байт идут на заголовки блока и записи). большие файлы могут потребовать нескольких косвенных записей

(различаемых по полю offset в ключе) плюс “хвост”.

Журнал

Журнал в reiserfs имеет фиксированный размер: для реализации в Linux2.4.x – это 8192 блоков + 1 блок для заголовка журнала. Журнал состоит из транзакций переменной длины и заголовка журнала. Он начинается списком транзакций, а заканчивается заголовком. Транзакция охватывает по крайней мере 3 дисковых блока и заголовок журнала, занимающий в точности один блок. Журнал реализован в виде кольцевого буфера.

Принято считать, что reiserfs журналирует только метаданные. Это не совсем верно. Не смотря на то, что целью журналирования является сохранение целостности метаданных, reiserfs журналирует некоторые дисковые блоки, т.к. они должны появиться в файловой системе лишь после успешной модификации метаданных, и записывать их прямо на диск до завершения транзакции нельзя. Таким образом каталоги, stat data и маленькие файлы, хранящиеся прямо в листьях дерева, также могут попасть в журнал и будут использованы для реконструкции файловой системы.

Заголовок журнала

Заголовок журнала – целый блок, описывающий, где в журнале находится первая несброшенная (unflushed) транзакция. Он занимает последний блок в журнале и содержит только 12 байт данных – остальная часть блока неопределена.

См. include/linux/reiserfs_fs.h

```
struct reiserfs_journal_header {
 __le32 j_last_flush_trans_id; /* ID последней удачно завершенной транзакции */
 __le32 j_first_unflushed_offset; /* Смещение (в блоках) следующей транзакции
       в журнале (с нее будет начато восстановление ФС в случае сбоя) */
 __le32 j_mount_id; /* Mount ID последней завершенной транзакции */
 struct journal_params jh_journal;
};
```

Структура jh_journal, очевидно, хранит копию параметров журнала из суперблока.

Транзакция, на которую указывает поле offset, должна иметь больший trans ID или mount ID, чем завершенная транзакция, что бы быть обработанной как несброшенная. В противном случае все транзакции объявляются завершенными, а блок, адресуемый по offset, используется для начала записи новой транзакции.

Транзакции

Транзакции описывают изменения в файловой системе. Вместо того, что бы напрямую записывать измененные блоки в дерево файловой системы, reiserfs сначала вносит их в журнал, отображая при этом на реальное местоположение в файловой системе.

Транзакция состоит из дескриптора, массива блоков и завершающего блока в конце. Все эти блоки смежны в пределах журнала.

Дескриптор транзакции

См. include/linux/reiser_fs_fs.h

```
struct reiserfs_journal_desc {
 __le32 j_trans_id; /* ID транзакции */
 __le32 j_len; /* Длина транзакции в блоках. len+1 – завершающий блок */
 __le32 j_mount_id; /* Mount ID транзакции */
 __le32 j_realblock[1]; /* Отображение на реальные координаты для каждого блока */
};
```

В конце дескриптора также записывается magic string – "ReIsErLB".

Отображение блоков означает следующее: поле j_realblock представляет собой массив, содержащий для каждого блока транзакции номер соотвествующего блока файловой системы. Если места в дескрипторе транзакции не хватает для отображения всех ее блоков, дополнительно используется завершающий блок. Таким образом, максимальный размер транзакции ограничен 2*(blocksize-21)/4 блоками, однако в действительност этот предел установлен в суперблоке.

Завершающий блок

Завершающий транзакцию блок содержит копию ID транзакции и ее длину.

См. include/linux/reiser_fs_fs.h

```
struct reiserfs_journal_commit {
 __le32 j_trans_id; /* Transaction ID */
 __le32 j_len; /* Длина транзакции */
 __le32 j_realblock[1];
};
```

Аналогичные по тематике переводы и статьи можно найти на fresco.front.ru.