author: Пешеходов Андрей aka fresco (filesystems@nm.ru)
released: 11.06.2008
modified: 25.08.2008

Статья была опубликована в журнале "Системный администратор", № 6 (июнь) 2008 года.

# Архитектура и реализация btrfs

Файловая система ZFS от Sun Microsystems, вышедшая в 2005 году, явилась настоящим прорывом в области архитектуры универсальных файловых систем общего назначения. Однако цели и задачи, которые ставят перед собой разработчики новой файловой системы btrfs от компании Oracle, впечатляют даже после знакомства с особенностями ZFS, которую, казалось бы, никто не сможет превзойти еще много лет.

Btrfs, о начале разработки которой было объявлено в июне 2007 года, находится сейчас на стадии альфа-версии. Тем не менее, основные ее возможности ясны, и многие из них можно попробовать уже сейчас.

Посмотрим подробнее, что именно предлагает пользователям Chris Mason — основной разработчик btrfs:

- Поддержка доступных на запись снапшотов (аналог клонов ZFS).
- Поддержка субтомов — множественных именованных корней в одной файловой системе с общим пулом хранения.
- Поддержка сложных многодисковых конфигураций — RAID уровней 0, 1, 5, 6 и 10, а также реализация различных политик избыточности на уровне объектов ФС — то есть возможно назначить, к примеру, зеркалирование для какого-либо каталога или файла.
- Copy-on-write (COW-) журналирование.
- Контроль целостности блоков данных и метаданных с помощью контрольных сумм.
- Зеркалирование метаданных даже в однодисковой конфигурации.
- Полностью распределенное блокирование. Давно известно, что все составные объекты множественного доступа, защищаемые глобальной блокировкой, имеют серьезные проблемы с производительностью. В btrfs распределенное блокирование реализовано для верхних уровней всех B-деревьев. Алгоритмы обработки метаданных продуманы таким образом, что бы не удерживать блокировки на разделяемых данных во время ожидания ввода/вывода. В ZFS этому вопросу также уделено достаточно серьезное внимание.
- Поддержка ACL.
- Защита от потери данных.
- Выбор хэш-алгоритма.
- Поддержка NFS.
- Флаги совместимости, необходимые для изменения дискового формата в новых версиях btrfs с сохранением совместимости со старыми.
- Резервные копии суперблока, по крайней мере — по одной на устройство.

- Скоростные приоритеты для дисков. Диск-итемы btrfs (своего рода дескрипторы дисков пула; их роль и структура будут показаны ниже) имеют поля для хранения показателей производительности устройства. Эти счетчики учитываются при выборе устройства для размещения данных того или иного типа. Аллокатор также стремиться выбирать наименее занятый диск для равномерного распределения нагрузки по всему пулу.
- Гибридные пулы. btrfs старается перемещать наиболее используемые данные на самое быстрое устройство, вытесняя с него "залежавшиеся" блоки. Эта политика хорошо согласуется с появившейся недавно моделью использования SSD (Solid State Drive). В частности, компания Sun Microsystems планирует в 2008 году начать выпуск серверов с небольшими SDD, используемыми в качестве быстрой памяти для наиболее "популярных" данных. К слову сказать, в ZFS возможности приоретизации дискового трафика не предусмотрено, поэтому Sun вынуждена будет реализовывать ее на более высоких уровнях, теряя универсальность такого решения. Справедливости ради стоит заметить, что поддержка приоритетов устройств может быть относительно легко добавлена и в ZFS — правда, ценой изменения дискового формата.
- Directory-операции с деревом корней. Дерево корней в btrfs хранит глобальную информацию о субтомах и снапшотах пула, а также о других группах метаданных. Со временем дерево корней станет настоящим каталогом и будет поддерживать все характерные системные вызовы.
- Балансировка данных между устройствами в btrfs возможна сразу после добавления диска к пулу, отдельной командой — а не только постепенно, в процессе использования (как это реализовано в ZFS).
- Диски для горячей замены, поддержка которых появилась и в ZFS.
- Он-лайн конфигурирование RAID будет реализовано на уровне объектов файловой системы — субтомов, снапшотов, файлов. Возможно будет также устанавливать некоторые параметры ввода-вывода для каталогов — с наследованием этих свойств всеми дочерними объектами.
- Выделение и резервирование objectid. В настоящее время вновь выделенный objectid (аналог номера inode) представляет собой число, равное последнему objectid+1. Более оптимально будет резервирование диапазона objectid каждым каталогом, для чего планируется вести специальный итем (итемом здесь называется объект B-дерева, инкапсулирующий какие-либо сторонние, не относящиеся к самому дереву данные). Код выделения inode также должен искать "дыры" в пространстве ключей, а не просто инкрементировать последний.
- Производительность вызова fsync(), фиксирующего на диск все грязные данные, является для btrfs большей проблемой, нежели для других ФС, т.к. объем побочного дискового трафика (за счет множества метаданных, разбросанных по разным B-деревьям) достаточно велик. Путем решения проблемы видится создание своеобразного журнала логических операций и частичный сброс только пользовательских данных. При частичной фиксации вызов fsync() будет сбрасывать на диск изменившиеся части только одного дерева, занося информацию об остальных модификациях в особый итем дерева корней. Смонтированная после крэша ФС будет считывать эти данные и вносить оставшиеся изменения уже во время

эксплуатации пула, не заметно для пользователя.
- Конвертер из ext2/3/4.

**Дизайн**

Btrfs реализована на простых и хорошо известных механизмах. Все они должны давать хороший результат и сразу после mkfs, однако более важным разработчики сочли сохранение хорошей производительности на старой, интенсивно используемой файловой системе.

Btrfs, как и подавляющее большинство современных файловых систем, начинается с суперблока, отстоящего на 16 Kb от начала диска. Его структура описана в ctree.h:

```
/*
 * Суперблок btrfs – по сути, список всех B-деревьев пула.
 */
struct btrfs_super_block {
      u8 csum[BTRFS_CSUM_SIZE];     /* контрольная сумма */
      u8 fsid[16];                  /* UUID файловой системы */
      __le64 bytenr;                /* номер этого блока */
      __le64 flags;

      __le64 magic;           /* 8 байт, "_B5RfS_M" */
      __le64 generation;      /* ID последней удачной транзакции */
      __le64 root;            /* указатель на дерево корней */
      __le64 chunk_root;      /* указатель на дерево сегментов */
      __le64 total_bytes;     /* размер ФС в байтах */
      __le64 bytes_used;      /* использовано байт */
      __le64 root_dir_objectid;    /* objectid корневого каталога */
      __le64 num_devices;     /* количество устройств в пуле */
      __le32 sectorsize;      /* размер сектора */
      __le32 nodesize;        /* размер узла дерева */
      __le32 leafsize;        /* размер листа дерева */
      __le32 stripesize;
      __le32 sys_chunk_array_size;
      u8 root_level;          /* глубина основного дерева ФС */
      u8 chunk_root_level;    /* глубина дерева сегментов */
      struct btrfs_dev_item dev_item;     /* дескриптор этого устройства */
      char label[BTRFS_LABEL_SIZE];       /* символьная метка ФС */
      u8 sys_chunk_array[BTRFS_SYSTEM_CHUNK_ARRAY_SIZE];
};
```

Диски, принадлежащие файловой системе, описываются dev-итемами, которые, в настоящее время, является частью суперблока, а  в будущем будут вынесены в отдельное B-дерево, продублированное на каждом из дисков:

```
struct btrfs_dev_item {
      __le64 devid;           /* ID устройства */
      __le64 total_bytes;     /* размер устройства */
      __le64 bytes_used;      /* использовано байт */
      __le32 io_align;        /* оптимальное выравнивание */
      __le32 io_width;        /* оптимальная пропускная способность */
      __le32 sector_size;     /* размер сектора дика */
```

```
      __le64 type;            /* тип устройства */
      __le32 dev_group;       /* информация о группе */
      u8 seek_speed;          /* скорость перемещения головки диска */
      u8 bandwidth;           /* максимальная полоса пропускания */
      u8 uuid[BTRFS_UUID_SIZE];    /* UUID файловой системы,которой
                                    * принадлежит диск */
};
```

Все остальные метаданные btrfs являются частью какого-либо B-дерева. Причем количество деревьев гораздо меньше количества типов метаданных — то есть во многих из них хранятся объекты различных, но логически связанных классов. Это напоминает архитектуру reiserfs и reiser4 — когда вся файловая система, по сути, является одним большим B-деревом, состоящем из данных и метаданных всех разновидностей. Как будет показано далее, это далеко не последнее сходство btrfs с файловыми система компании NameSys.

**Структура B-дерева**

Реализация B-дерева btrfs (это классическое B+ дерево с данными в листьях и указателями в узлах) обеспечивает базовую функциональность для эффективного хранения и поиска большого количества типов данных. Btree-код знает только о трех структурах: ключи, итемы  и заголовки блоков (см. ctree.h):

```
/*
 * Каждый блок дерева (листовой или внутренний) начинается
 * с заголовка блока
 */
struct btrfs_header {
      u8 csum[BTRFS_CSUM_SIZE];    /* контрольная сумма блока */
      u8 fsid[BTRFS_FSID_SIZE];    /* UUID файловой системы, владеющей
                                    * блоком */
      __le64 bytenr;               /* адрес блока на диске */
      __le64 flags;
      u8 chunk_tree_uuid[BTRFS_UUID_SIZE];     /* UUID дерева сегментов */
      __le64 generation;           /* ID транзакции */
      __le64 owner;                /* ссылка на родителя блока */
      __le32 nritems;              /* количество итемов в блоке */
      u8 level;                    /* уровень блока в дереве */
};

/* Ключ */
struct btrfs_key {
      u64 objectid;
      u8 type;
      u64 offset;
};

/* Итем */
struct btrfs_item {
      struct btrfs_disk_key key;   /* Ключ */
      __le32 offset;               /* Смещение пакета данных в листе */
      __le32 size;                 /* Длина пакета данных */
};
```

Внутренние узлы дерева содержат только пары ключ-итем, листья же разбиты на две секции, растущие к середине узла. В начале листового узла хранятся итемы фиксированного размера, в конце — данные этих итемов. Данные итема интерпретируются на более высоких уровнях согласно полю type соответствующего ключа. См. ctree.h:

```
/* Итем btrfs */
struct btrfs_item {
      struct btrfs_disk_key key;    /* Ключ */
      __le32 offset;                /* Смещение данных */
      __le32 size;                  /* Размер */
};

/* Листовой узел B-дерева btrfs */
struct btrfs_leaf {
      struct btrfs_header header;   /* Заголовок блока */
      struct btrfs_item items[];    /* Массив итемов */
};

/* Пара ключ-указатель — элемент внутреннего узла B-дерева btrfs */
struct btrfs_key_ptr {
      struct btrfs_disk_key key;    /* Ключ */
      __le64 blockptr;              /* Указатель на потмка */
      __le64 generation;            /* ID транзакции */
} __attribute__ ((__packed__));

/* Внутренний узел B-дерева btrfs */
struct btrfs_node {
      struct btrfs_header header;   /* Заголовок блока */
      struct btrfs_key_ptr ptrs[];  /* Массив пар ключ-указатель */
} __attribute__ ((__packed__));
```

Заголовок блока дерева содержит контрольную сумму содержимого блока, UUID (Universally Unique Identifier — универсальный уникальный идентификатор стандарта OSF DCE) файловой системы, которой принадлежит блок, уровень блока в дереве и смещение, по которому блок располагается на диске. Эти поля позволяют проверить целостность метаданных при чтении. В будущем планируется также хранить здесь 64-битный sequence-номер, которые будет содержаться также и в родительском для данного блока узле. Это позволит файловой системе обнаруживать и исправлять фантомные записи на диск (когда блок пишется по ошибочным координатам). Заметим, что использовать для этих целей контрольную сумму дочернего узла нельзя, т.к. она не хранится в его родителе для упрощения отката транзакции. Sequence-номер будет эквивалентен времени вставки блока в дерево, в то время как контрольная сумма вычисляется еще до размещения блока и, к тому же, меняется при каждой модификации блока.

Поле generation соответствует ID транзакции, в рамках которой был размещен или переразмещен данный блок. Оно позволяет легко поддерживать инкреметальные бэкапы (снапшоты) и подсистему COW-транзакций — совершенно так же, как это сделано в ZFS.

**Структуры данных файловой системы**

Каждый объект файловой системы имеет objectid, динамически выделяемый при его создании.

Поле offset ключа хранит логическое смещение данных в пределах описываемого объекта. Например, для файловых экстентов это будет смещение экстента от начала файла. Поле type содержит идентификатор типа итема, а также зарезервированное пространство для расширения в будущем.

**Inodes**

Inodes хранятся в структуре btrfs_inode_item (ctree.h), поле offset ключа inode-итема всегда равно нулю, поле type — единице. Таким образом, математически, ключ inode-итема является наименьшим среди всех итемов данного объекта. Inode-итем хранит традиционные stat-данные:

```
struct btrfs_inode_item {
      __le64 generation;      /* ID транзакции создания файла */
      __le64 size;            /* Размер файла в байтах */
      __le64 nblocks;         /* Количество занимаемых блоков */
      __le64 block_group;     /* Предпочитаемая группа блоков */
      __le32 nlink;           /* Счетчик ссылок на файл */
      __le32 uid;             /* UID владельца */
      __le32 gid;             /* GID владельца */
      __le32 mode;            /* Маска типа и прав доступа */
      __le64 rdev;            /* [minor:major] для устройств */
      __le16 flags;           /* Флаги */
      __le16 compat_flags;    /* Флаги совместимости */
      struct btrfs_timespec atime;  /* Времена доступа, модификации, и т.д. */
      struct btrfs_timespec ctime;
      struct btrfs_timespec mtime;
      struct btrfs_timespec otime;
};
```

Поле compat_flags введено для реализации совместимости со старыми версиями файловой системы. Те или иные биты отведены под флаги версии во всех структурах данных btrfs для того, что бы безопасно изменять дисковый формат ФС даже после официального релиза.

**Файлы**

Содержимое маленьких файлов (размером не более блока) может храниться прямо в B-дереве, в данных экстент-итема. В этом случае поле offset ключа экстент-итема хранит смещение данных внутри файла, а поле size структуры btrfs_item показывает, сколько места в листе занимает данный итем. Таких своеобразных экстент-итемов может быть несколько на файл.

Большие файлы хранятся в экстентах. Структура btrfs_file_extent_item содержит ID транзакции размещения экстента (поле generation) и пару [смещение,длина], описывающую его положение на диске. Эскстент также хранит логическиеы смещение и длину в уже существующем экстенте. Это позволяет btrfs

безопасно писать в средину длинного экстента без предварительного перечитывания старых данных файла (относящихся к предыдущему снапшоту, к примеру).

```
struct btrfs_file_extent_item {
      __le64 generation;
      u8 type;
      /*
       * Дисковое пространство, используемое экстентом. Блоки
       * конторольных сумм включены
       */
      __le64 disk_bytenr;
      __le64 disk_num_bytes;

      /*
       * Логиическое смещение данного экстента в файле (без учета
       * блоков контрольных сумм). Это позволяет экстент-итему
       * указывать в середину существующего экстента, разделяя его
       * между двумя снапшотами (если в новом снапшоте изменились данные).
       */
      __le64 offset;

      /* Логическое количество блоков (без учета контрольных сумм)*/
      __le64 num_bytes;
};
```

Контрольные суммы данных файла хранятся в B-дереве в csum-итеме с соответствующим objectid (структура btrfs_csum_item, ctree.h). Поле offset ключа csum-итема указывает на смещение защищаемого участка данных от начала файла. Один такой итем может хранить несколько контрольных сумм. Csum-итем используется только для файловых (больших) экстентов, встроенные в дерево маленькие файлы защищаются контрольной суммой в заголовке блока. Если csum-итем для некоторого пакета данных не представлен, пакет считается не инициализированным — при чтении возвращается блок нулей (в будущем поведение в этом случае будет выбирается пользователем при создании ФС — может возвращаться также код ошибки EIO).

**Каталоги**

Каталоги btrfs индексируются двумя способами. Для поиска по имени файла используется индекс, составленный из objectid каталога, константы BTRFS_DIR_ITEM_KEY и 64-битного хэша имени. По умолчанию используется TEA-хэш, но могут быть добавлены и другие алгоритмы (определяется по полю flags в inode каталога). Второй способ индексирования используется вызовом readdir(), возвращающем данные в порядке возрастания номеров inodes, приближенном к порядку следования блоков на диске (согласно принятой политике размещения). Этот способ дает большую производительность при чтении данных большими пакетами (бэкапы, копирование, и т.д.), а также позволяет быстро проверить линковку inode с каталогом (подсчет количества ссылок на файл при fsck). Этот индекс состоит из objectid каталога, константы BTRFS_DIR_INDEX_KEY и inode objectid.

### Учет ссылок на экстенты

Учет ссылок на объекты — основа любой файловой системы с поддержкой снапшотов. Для каждого экстента, выделенного дереву или файлу, btrfs записывает количество ссылок в структуре btrfs_extent_item. Деревья, хранящие эту информацию, служат также картами выделенных экстентов файловой системы. Некоторые деревья не поддерживают учета ссылок и защищаются только COW-журналированием. Однако структура экстент-итемов одинаков для всех выделенных блоков.

### Группы блоков

Группы блоков позволяют оптимизировать аллокатор путем разбиения диска на участки длиной от 256 Mb. Для каждого участка доступна информация о свободных блоках. Поле block_group каждого inode хранит номер предпочитаемой группы блоков, в которой btrfs будет стараться разместить новые данные объекта. Группа блоков — просто надстройка над сегментом (см. ниже), позволяющая быстро оценить количество свободного пространства и тип (данные/метаданные) без обращения к дереву сегментов, которое происходит, когда найден сегмент, располагающий достаточным пространством и подходящим типом.

```
struct btrfs_block_group_item {
      __le64 used;            /* использовано блоков */
      __le64 chunk_objectid;  /* objectid соответствующего сегмента */
      __le64 flags;           /* флаги, в том числе тип (данные/метаданные) */
}
```

Группа блоков имеет флаг, показывающий, данные или метаданные она хранит. При создании ФС 33% групп блоков выделяются под метаданные, 66% — под данные. При заполнении диска это предпочтение может быть пересмотрено, однако, в любом случае, btrfs старается избежать смешивания данных и метаданных в одной группе блоков. Это решение существенно улучшает производительность fsck и уменьшает количество перемещений головок диска при отложенной записи ценой небольшого увеличения количества перемещений при чтении.

### Деревья свободных экстентов

Деревья свободных экстентов, в том числе, служат btrfs для разбиения доступного дискового пространства на участки с различными политиками выделения блоков. Каждое дерево экстентов владеет сегментом указанного диска, блоки которого могут быть выделены объектам различных субтомов. Политики будут определять, каким образом распределять данные по доступным деревьям экстентов, позволяя пользователю назначать зеркалирование, распределение данных (strippping) или квотирование различных частей диска.

Btrfs будет интегрирована с менеджером дисков для упрощения управления

большими пулами хранения. Основная идея состоит в назначении по крайней мере одного дерева экстентов для каждого диска для предоставления пользователю возможности назначать их субтомам, каталогам или файлам.

**Обратные ссылки**

Обратные ссылки в btrfs служат для:

- Учета всех владельцев ссылки на экстент для корректного его освобождения.
- Обеспечения информации для быстрого поиска ссылающихся на данный экстент объектов, если некоторый блок нуждается в исправлении или переразмещении.
- Упрощения перемещения блока при урезании ФС и других операциях управления пулом.

**Обратные ссылки на файловые экстенты**

На файловые экстенты могут ссылать следующие объекты:

- Снапшоты, субтома и различные их поколения
- Разные файлы внутри одного субтома
- Разные логические экстенты внутри одного файла

Структура обратной ссылки такова (ctree.h):

```
struct btrfs_extent_ref {
      __le64 root;            /* objectid корня субтома */
      __le64 generation;      /* номер поколения дерева, владеющего ссылкой */
      __le64 objectid;        /* objectid файла, владеющего ссылкой */
      __le64 offset;          /* смещение в файле */
};
```

При выделении файлового экстента эти поля заполняются следующей информацией: objectid корня субтома, id транзакции, inode objectid и смещение в файле. При захвате ссылки на лист новая обратная ссылка добавляется для каждого файлового экстента. Это похоже на создание экстента, однако поле generation инициализируется идентификатором текущей транзакции.

При удалении файлового экстента или некоторого снапшота, находится и удаляется соответствующая обратная ссылка.

**Обратные ссылки на btree-экстенты**

Ссылки на btree-экстенты могут захватывать следующие объекты:

- Различные сбутома
- Разные поколения одного субтома

Хранение всеобъемлющей информации для полноценного обратного отображения потребовало бы хранения наименьшего ключа данного листового блока в обратной ссылке. Это не удобно, т.к. при каждой модификации (например, изменении поля offset, что происходит достаточно часто) этого ключа пришлось бы модифицировать и ссылку.

Вместо этого btrfs хранит только objectid наименьшего ключа на том же уровне, что и данный блок. Поиск по дереву останавливается на уровень выше, чем записано в обратной ссылке.

В некоторых деревья btrfs учет обратных ссылок не ведется: например, в деревьях экстентов и корней. Обратные ссылки в этих деревьях всегда имеют поле generation=0.

При размещении блока дерева создается такая обратная ссылка:

| objectid корня субтома | id транзакции или ноль | уровень узла | наименьший objecid узла |
|---|---|---|---|

Уровень хранится в поле objectid структуры btrfs_extent_ref, т.к. максимальный уровень равен 255, а минимальный objectid 256. Таким образом btrfs отличает файловые обратные ссылки от btree-ссылок.

Если ссылка на блок захватывается неким объектом, в дерево также вставляется обратная ссылка с соответствующими данными о владельце и транзакции.

**Построение ключа обратной ссылки**

Обратная ссылка имеет 4 64-битных поля, которые хэшируются в единственно 64-битное значение, что помещается в поле offset ключа. Поле objectid ключа соответствует ID описываемого объекта, а поле type инициализируется константой BTRFS_EXTENT_REF_KEY.

**Снапшоты и субтома**

Субтома представляют собой именованные B-деревья, содержащие иерархию файлов и каталогов, и имеют inodes в дереве корней. Субтом может быть ограничен квотой на количество блоков; на все блоки и файловые экстенты, принадлежащие субтому, ведется учет ссылок для поддержки снапшотов. Предельное количество субтомов в файловой систмеме btrfs — 2^64.

Снапшоты по внутренней структуре идентичны субтомам, однако их корневой блок изначально разделяется с другим (родительским) субтомом. Когда снапшот создан, файловая система увеличивает количество ссылок на корневой блок, и далее подсистема COW-транзакций фиксирует изменения, сделанные в корневых блоках сбутома и снапшота, уже в разных местах. Снапшоты btrfs доступны на

запись и бесконечно рекурсивны. При необходимости создания read-only снапшота его блочная квота устанавливается в единицу сразу при инициализации.

**Корни B-деревьев**

Каждая файловая система формата btrfs состоит из нескольких корней B-деревьев. Только что созданная ФС имеет корни для:

- Дерева корней
- Дерева выделенных экстентов
- Дерева default-субтома

Дерево корней содержит корневые блоки для дерева экстентов, а также корневые блоки и имена деревьев для каждого субтома и снапшота в ФС. При фиксации транзакции указатели на корневые блоки обновляются по COW-семантике в этом дереве, и его новый корневой блок записывается в суперблок btrfs.

Дерево корней организовано в виде каталога всех других деревьев файловой системы и имеет directiry-итемы для хранения имен снапшотов и субтомов. Каждый субтом имеет objectid в этом дереве и не менее одной структуры btrfs_root_item. Directory-итемы отображают имена субтомов на их root-итемы Т.к. ключ root-итема обновляется на каждой транзакции, directory-итем ссылается на номер поколения, что позволяет всегда найти наиболее новую версию какого-либо корня. Структура root-итема (ctree.h):

```
struct btrfs_root_item {
      struct btrfs_inode_item inode;      /* inode-дескриптор */
      __le64 root_dirid;      /* objectid основного дерева субтома */
      __le64 bytenr;          /* размер */
      __le64 byte_limit;      /* квота */
      __le64 bytes_used;      /* использовано байт */
      __le32 flags;
      __le32 refs;            /* количество ссылок */
      struct btrfs_disk_key drop_progress;
      u8 drop_level;
      u8 level;
};
```

Деревья свободных экстентов используются для управления выделением дискового пространства. Доступное место может быть разделено между несколькими деревьями экстентов для уменьшения влияния блокировок и реализации различных политик выделения для разных участков диска.

Суперблок btrfs указывает на дерево корней, которое, в свою очередь, содержит указатели на деревья свободных экстентов и субтомов (субтома хранятся в root-итемах). Дерево корней также имеет указатель на каталог, отображающий имена субтомов на root-итемы в дереве корней. Показанная файловая система имеет один субтом с именем "default" и один его снапшот с именем "snap".

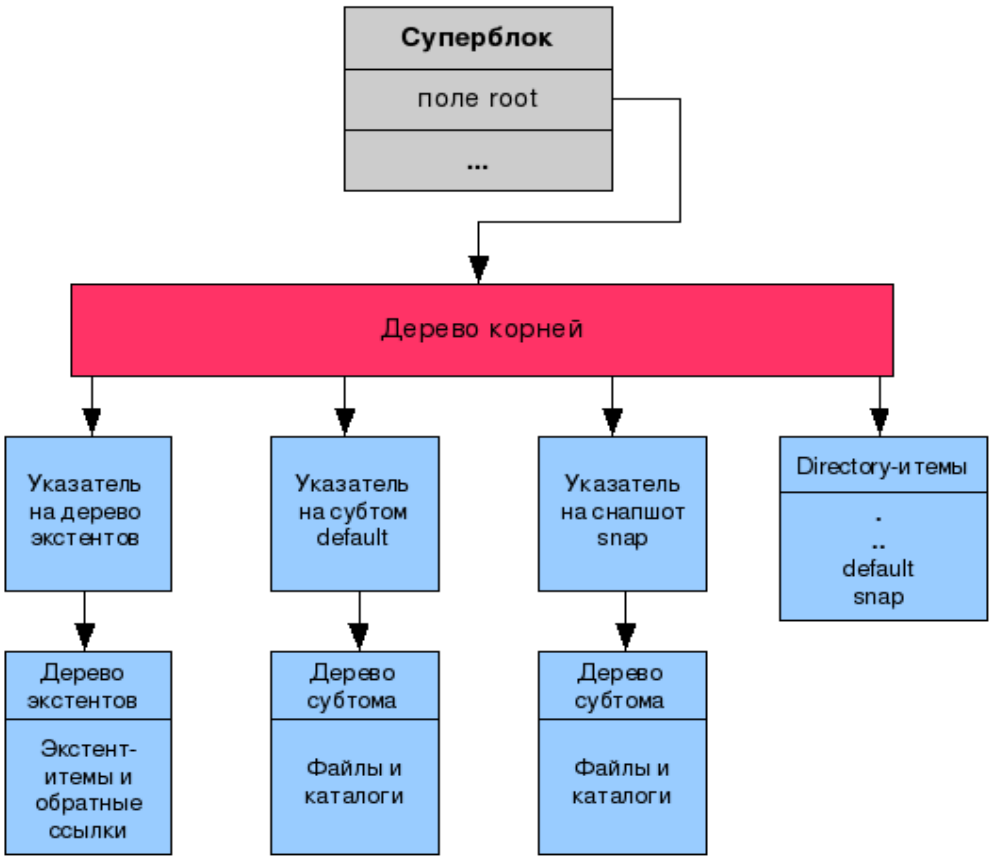


**Рис. 1. B-деревья btrfs**

### Многодисковые конфигурации

Btrfs, как и основной ее аналог — ZFS — поддерживает организацию сложных пулов хранения из нескольких дисков. Основные ее возможности в этой области таковы:

- Зеркалирование метаданных в конфигурации до N зеркал (N>2)
- Зеркалирование метаданных на одном устройстве
- Зеркалирование экстентов данных
- Обнаружение ошибок записи с помощью контрольных сумм и их коррекция из зеркальной копии
- Распределенные (stripped) экстенты данных
- Различные политики зеркалирования на одном устройстве
- Эффективное перемещение данных между устройствами
- Эффективное переконфигурирование хранилища
- Динамическое выделение пространства для каждого субтома

Если бы btrfs полагалась на device mapper или MD для реализации поддержки многодисковых конфигураций, она потеряла бы большинство из своих сильных сторон: обработку и корректировку ошибок записи с помощью контрольных сумм, перемещение данных между устройствами и, как следствие, возможность изменения размера тома, гибкие политики выделения, зеркалирование метаданных даже на единственном диске. Именно поэтому в btrfs, как и в ZFS, реализован

собственный уровень объединения устройств, не полагающийся на уже существующие в Linux программные RAID-системы.

В настоящее время btrfs поддерживает конфигурации RAID0, RAID1 и RAID10, реализация RAID5 и RAID6 запланирована.

**Сегменты (storage chunks)**

Сегментом btrfs называется обособленный участок диска с логической адресацией. Все указатели на экстенты работают с сегментными адресами вместо физических дисковых. Суперблок имеет особую секцию, отображающую сегменты на дисковые адреса через дерево сегментов. Код сегментации — единственная часть btrfs, имеющая дело с физическими адресами. Весь остальной драйвер работает с сегментными.

Каждый сегмент располагает пространством, выделенным с одного или нескольких устройств, для реализации зеркала или распределенного хранилища (stripe). Минимальный размер сегмента btrfs равен 256 Mb, средний — 1/100 объема устройства.

Каждый сегмент располагает единственным деревом выделенных экстентов и имеет обратную ссылку на это дерево.

**Разрешение сегментных адресов**

Каждому устройству, добавляемому к файловой системе, назначается 64-битный идентификатор (device ID). Информация обо всех устройствах пула отсортирована по device ID в особом B-дереве. Каждый корень дерева в ФС связан с единственным деревом сегментов для разрешения сегментных адресов. ID сегмента продублирован в каждом блоке дерева, поэтому может использоваться во время fsck.

**Размещение сегментов**

Каждое устройство, добавленное в пул, имеет дерево размещения, отслеживающее, какая область диска какому сегменту назначена. Обратные ссылки в этом дереве отслеживают, какой сегмент размещен в какой части устройства. Так как экстентов на устройство приходится относительно немного, это дерево разделяется несколькими дисками.

Сегменты назначаются некоторому дереву выделения экстентов и используются для разрешения запросов на размещение экстентов для данных и метаданных. При расширении файловой системы, сегменты добавляются в дерево динамически. Дерево размещения экстентов осуществляет выделения пространства в сегментах, отслеживает свободное место в них и обратные ссылки на связанные экстенты других сегментов.

**Управление сегментами**

Логическая адресация позволяет достаточно просто перемещать сегменты. Дерево выделенных экстентов, владеющее данным сегментом, располагает информацией о занятых/свободных участках сегмента, и может гибко и эффективно управлять копированием только необходимых данных.

Устройства в файловой системе могут быть удалены или сбалансированы благодаря перемещению сегментов. На вновь добавленное устройство могут быть перенесены существующие сегменты с других дисков (для балансировки нагрузки). Диск может использоваться и только для новых размещений.

Восстановление зеркальных пулов осуществляется путем обхода дерева выделенных экстентов, проверки наличия недоступных дисков и следования по обратным ссылкам размещенных на них сегментов. Каждый сегмент исправляется индивидуально, так что можно ограничить исправление обходом только тех сегментов, которые действительно используются.

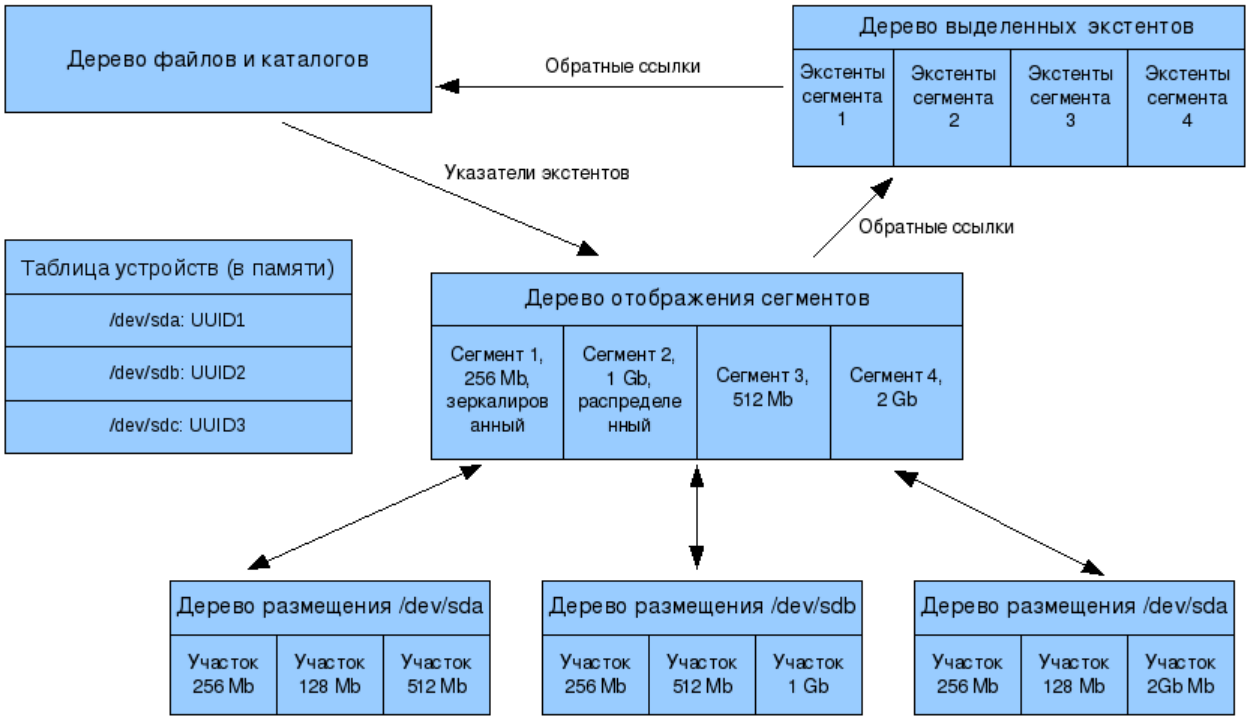


**Рис. 2. Трансляция сегментных адресов.**

### Разрешение ID дисков

Идентификатор устройства хранится в суперблоке диска. Устройства сканируются утилитой btrfsctl для построения списка дисков, назначенных файловой системе с данным UUID. Деревья сегментов также хранят информацию о каждом из устройств, так что корректность списка может быть проверена во время монтирования.

### Btrfs для администратора

Попробуем btrfs в работе. Исходные тексты модуля ядра и утилит стабильной версии доступны по ссылке [2]. Модуль ядра собираем и устанавливаем так (не забываем, что он зависит от CONFIG_LIBCRC32C; заголовки ядра также должны быть доступны по пути /lib/modules/`uname -r`/build):

```
make
make install
```

Набор утилит btrfs-progs использует libuuid, которая входит в состав e2fsprogs — то есть при сборке нужны соответствующие заголовки. В дистрибутивах необходимый пакет обычно называется e2fsprogs-devel или libuuid-devel. Сборка btrfs-progs производится также вполне привычно:

```
make
make install
```

Теперь попробуем создать и смонтировать файловую систему на одном диске — здесь все вполне традиционно:

```
modprobe btrfs
mkfs.btrfs /dev/sda1
```

mkfs.btrfs можно использовать с опциями:

- -b, --byte-count — задает размер файловой системы
- -l, --leafsize — задает размер листового узла дерева
- -n, --nodesize — задает размер внутреннего узла дерева
- -s, --sectorsize — задает размер минимального выделяемого элемента (не менее физического сектора диска)

Монтируем субтом default (создается по умолчанию):

```
mount -t btrfs /dev/sda1 /mnt/test
```

Можно смонтировать не какой-либо субтом, а все дерево корней:

```
mount -t btrfs /dev/sda1 /mnt/test -o subvol=.
```

Тогда содержимое субтома default окажется в папке /mnt/test/default. Создать новый субтом можно командой:

```
btrfsctl -s new_subvol /mnt/test
```

Видим что, теперь в папке /mnt/test 2 элемента: default и new_subvol.

Посмотрим, как btrfs создает снапшоты:

```
btrfsctl -s new_subvol_snap /mnt/test/new_subvol
```

Также с помощью btrfsctl можно поменять размер файловой системы (если она еще не занимает весь диск или в пул было добавлено новое устройство):

```
btrfsctl -r +4g /mnt/test
```

btrfs версии 0.14 — первый релиз с поддержкой многодисковых пулов. К существующей файловой системе можно добавлять устройства, но пока нельзя удалять (пока также не понятно, как удалять снапшоты). Обработка ошибок чтения или аппаратуры также еще толком не оттестирована, так что проверка работы btrfs в экстремальных условиях пока не возможна.

Так можно создать btrfs на двух дисках:

```
mkfs.btrfs /dev/sda1 /dev/sda2
```

Распределенный пул (stripe):

```
mkfs.btrfs -m raid0 /dev/sda1 /dev/sda2
```

Зеркало:

```
mkfs.btrfs -m raid1 /dev/sda1 /dev/sda2
```

Этой командой можно создать ФС, не зеркалируюущую метаданные на одном диске:

```
mkfs.btrfs -m single /dev/sda1
```

После создания btrfs команде mount можно передавать любой из дисков. Однако следует учитывать, что после выгрузки модуля btrfs.ko (после перезагрузки в том числе) необходимо пройтись по всем дисковым устройствам вашей системы командой:

```
btrfsctl -a
```

Или передать ей только используемые устройства:

```
btrfsctl -A /dev/sda1
```

Команда btrfs-show выдаст список всех обнаруженных файловых систем с их UUID и задействованными устройствами.

Добавить новый диск в пул можно так:

```
mount -t btrfs /dev/sda1 /mnt/test -o subvol=.
btrfs-vol -a /dev/sda3 /mnt/test
```

Файловую систему можно сбалансировать — то есть распределить часть данных и метаданных между существующими и вновь добавленным диском:

```
btrfs-vol -b /mnt/test
```

Также в btrfs-progs включена программа-конвертер файловой системы из ext3. В версии 0.16 пакета утилит код конвертера имеется, однако сборка его по умолчанию отключена, так как безопасно использовать его пока рано. Если риск возможной потери данных вас не пугает, собрать и использовать его можно так:

```
cd btrfs-progs
make convert
./btrfs-convert /dev/sda4
```

Простота реализации подобного конвертера основана на том, что btrfs почти не имеет метаданных с фиксированным дисковым положением (практически, это только суперблок). Более того, COW-семантика btrfs позволяет сохранить не тронутой оригинальную ext3 с возможностью отката на нее даже после внесения изменений в btrfs-копию.

Конвертер использует libe2fs для чтения метаданных ext3, и размещает метаданные btrfs только в свободных блоках оригинальной файловой системы. Работает он в такой последовательности:

1. Копирует первый мегабайт диска в "запас"
2. Читает каталоги и inodes, создает их копии в btrfs
3. Вставляет ссылки на блоки данных ext3 в метаданные btrfs

Первый мегабайт диска копируется в альтернативное местоположение, на его место записываются метаданные btrfs, остальные блоки, используемые ext3 (благодаря COW-журналированию) не перезаписываются. Откат обратно к ext3, очевидно, представляет собой просто восстановление первого мегабайта диска.

Конвертер создает снапшот, содержащий данные ext3 на момент конверсии. Блоки метаданных ext3 заносятся в отдельный файл, который может быть смонтирован как читаемая копия оригинальной ext3.

Смонтируем снапшот ext3:

```
mount -t btrfs /dev/sda4 /mnt/test -o subvol=ext2_saved
```

Смонтируем образ оригинальной ФС:

```
mount -t ext3 -o loop /mnt/test/image /mnt/ext3
```

Выполнить откат можно так:

```
umount /mnt/ext3
umount /mnt/test
btrfs-convert -r /dev/sda4
```

Если необходимости в откате больше нет, и нужно освободить место, занимаемое метаданными ext3, достаточно просто удалить файл image в снапшоте ext2_saved. Если есть желание избавиться от всех данных ext3 — удаляем сам снапшот.

**Ссылки**


1. Официальный сайт проекта: btrfs.wiki.kernel.org
2. Исходники btrfs: www.kernel.org/pub/linux/kernel/people/mason/btrfs
3. "Архитектура ZFS": www.filesystems.nm.ru/my/zfs_arch.pdf