

author: Пешеходов Андрей aka fresco (filesystems@nm.ru)
released: 19.01.2009
modified: 14.01.2009

Статья была опубликована в журнале "Системный администратор", № 11 (ноябрь) 2008 года.

XFS: структура и алгоритмы

Файловая система XFS разработки компании Silicon Graphics считается многими пользователями операционных систем Linux и IRIX одной из самых стабильных и производительных ФС поколения 90-х. Посмотрим, за счет каких архитектурных решений она добивается столь высоких показателей.

Историческая справка

Исследования концепций файловых систем, проводимые в начале 90-х годов прошлого века, предвещали скорое прекращение роста производительности подсистем ввода-вывода. Инженеры SGI также столкнулись с этим явлением. Проблемой стала не производительность аппаратуры, а ограничения, наложенные старой файловой системой ОС IRIX – EFS. EFS архитектурно подобна Berkley FFS, однако она использует экстенды вместо отдельных блоков для распределения дискового пространства и ввода-вывода. EFS была не в состоянии поддерживать файловые системы размером более 8 Gb, файлы длиннее двух гигабайт или предоставлять приложению канал ввода-вывода с производительностью на уровне пропускной способности аппаратуры. EFS не была спроектирована для работы на больших вычислительных системах, и ее возможностей уже в то время не хватало для того, что бы предоставить приложениям все возможности нового оборудования. Пока инженеры рассматривали направления возможного усовершенствования EFS, требования пользователей возросли настолько, что было решено заменить ее полностью новой файловой системой.

К XFS были предъявлены следующие требования:

- быстрое восстановление после сбоя
- поддержка больших разделов и файлов
- эффективная работа с большими каталогами
- хорошая масштабируемость как по производительности, так и по объемам хранилищ
- эффективное противодействие фрагментации
- высокая параллельность обработки запросов

Практически сразу стало очевидно, что создать ФС, удовлетворяющую всем этим требованиям, можно только с повсеместным применением B+ деревьев и учетом свободного пространства с помощью экстендов.

Проработка концепции XFS в целом завершилась к 1991 году, первые рабочие версии появились в SGI в 92-м. В 1994 году XFS вошла в релиз операционной системы IRIX 5.3.

В 2001, глядя на IBM, открывшую свою JFS, или же руководствуясь какими-то

иными соображениями, SGI открывает исходные код XFS под лицензией GPL v2 и начинает работы по формированию сообщества программистов и портированию ее в Linux. В мае 2001 года выходит релиз 1.0 этой ФС для Linux, через год код XFS принимается в экспериментальную ветку 2.5, а в декабре 2003 выходит ядро 2.6.0 со стабильной XFS на борту.

Архитектура

Файловая система XFS может включать в себя до трех секций: данных, журнала и реального времени. область данных содержит пользовательские данные и метаданные ФС, а также (опционально) зону журнала. Секцией реального времени называется некая обособленная часть дискового раздела, к которой применяются упрощенные механизмы доступа и выделения блоков и inodes.

Область данных делится на некоторое количество групп размещения (allocation group, AG), размер которых выбирается при создании ФС. AG's чем-то подобны группам блоков в ext2 или группам цилиндров в FFS, однако задуманы были не для удобства управления дисковым пространством, а для распараллеливания запросов к аллокаторам блоков и inodes.

В+ деревья

XFS использует В+ деревья повсюду. С их помощью индексируются пакеты inodes, списки свободных экстенгов, элементы каталогов и записи файловых карт. В+деревья XFS имеют вполне традиционную структуру: во внутренних узлах хранятся только ключи и указатели на потомков, а в листьях – ключи и данные. Так как типов деревьев в XFS существует несколько, общий код обрабатывает лишь стандартные заголовки блоков (xfs_btree.h):

```
/*
 * Комбинированный заголовок и структура, используемая
 * общим кодом
 */
typedef struct xfs_btree_hdr {
    __be32      bb_magic;      /* magic-номер, зависит от типа дерева
                               * и блока */
    __be16      bb_level;     /* уровень блока в дереве, 0 если лист */
    __be16      bb_numrecs;   /* текущее количество записей */
} xfs_btree_hdr_t;

/* Внутренний узел дерева */
typedef struct xfs_btree_block {
    xfs_btree_hdr_t  bb_h;      /* заголовок */
    union {

        /* указатели короткой формы */
        struct {
            __be32      bb_leftsib; /* сосед слева */
            __be32      bb_rightsib; /* сосед справа */
        } s;

        /* указатели длинной формы */
        struct {
            __be64      bb_leftsib; /* сосед слева */
            __be64      bb_rightsib; /* сосед справа */
        } l;
    } bb_u;
} xfs_btree_block_t;
```

Вслед за заголовком располагаются массивы данных. Для внутренних узлов это это 2 списка – ключей и указателей на потомков, растущие к середине. Для листьев – массив записей, отсортированный по возрастанию. Формат ключей и записей, соответственно, определяется типом дерева.

Группы размещения

Вообще говоря, XFS позиционируется как 64-битная файловая система. Однако драйвер старается избегать использования 64-битных указателей, пока в этом действительно нет нужды. Удержание указателей в рамках 32-битных значений является одним из мотивов применения `allocation groups`.

В среднем каждая AG имеет 0.5 – 4 Gb в размере и располагает собственными структурами данных для управления размещением `inodes` и блоков в своих пределах. Ограниченный размер AG's позволяет использовать внутри них относительные 32-битные номера `inodes`, что удерживает размеры структур данных в рамках оптимальных значений. Пока структуры обслуживают данные внутри своей AG, в них применяются относительные 32-битные номера, а глобальные структуры данных могут ссылаться на блоки и `inodes` в любом месте ФС с помощью 64-битных указателей.

AG лишь изредка применяются для группировки данных, вообще же они слишком велики для этого, а центрами сосредоточения данных в XFS (для уменьшения фрагментации и улучшения производительности чтения) служат файлы или каталоги.

Однако главная задача групп размещения – достижения явного параллелизма в управлении размещением блоков и `inodes`. Файловые системы предыдущего поколения имеют однопоточный механизм выделения и освобождения дискового пространства. На больших файловых системах с множеством использующих их процессов такой алгоритм может создать серьезные проблемы. Сделав структуры в каждой AG независимыми, разработчики XFS добились, что операции с дисковым пространством и `inodes` могут выполняться параллельно по всей ФС. Таким образом, несколько пользовательских процессов могут одновременно запросить свободное место или новый `inode`, и ни один из них не будет заблокирован (т. е. не будет ждать, пока с метаданными поработает другой процесс).

Первые 2 Кб любой группы размещения являются зоной метаданных. Она разбита на 4 функционально обособленных 512-байтных сектора. Первый сектор хранит копию суперблока, за ним следует AGF-блок, далее AGI- и AGFL-блоки.

Суперблок

Каждая AG хранит копию суперблока, однако только суперблок самой первой группы размещения (нулевой сектор дискового раздела) используется реально — остальные служат избыточной информацией для `xfs_repair` и необходимы в случае разрушения основного суперблока.

Структура суперблока такова (`xfs_sb.h`):

```
typedef struct xfs_sb {
```

```

__uint32_t sb_magicnum; /* magic-номер == XFS_SB_MAGIC ("XFSB") */
__uint32_t sb_blocksize; /* размер блока в байтах */
xfs_dfrsbno_t sb_dblocks; /* количество блоков данных */
xfs_dfrsbno_t sb_rblocks; /* количество realtime-блоков */
xfs_drtbno_t sb_rextents; /* количество realtime-экстентов */
uuid_t sb_uuid; /* идентификатор ФС */
xfs_dfsbno_t sb_logstart; /* стартовый блок журнала
 * (если он встроенный) */

xfs_ino_t sb_rootino; /* номер корневого inode */
xfs_ino_t sb_rbmino; /* inode, описывающий realtime-bitmap */
xfs_ino_t sb_rsumino; /* inode, описывающий realtime-summary */
xfs_agblock_t sb_rextsize; /* размер realtime-экстента в блоках */
xfs_agblock_t sb_agblocks; /* размер AG */
xfs_agnumber_t sb_agcount; /* количество AG's */
xfs_extlen_t sb_rmbmblocks; /* количество блоков в
 * realtime-bitmap */

xfs_extlen_t sb_logblocks; /* количество блоков в журнале */
__uint16_t sb_versionnum; /* битовая маска флагов ФС:
 * 0x0010 - используются атрибуты
 * 0x0020 - используются inodes версии 2
 * 0x0040 - используются квоты
 * 0x0080 - используется выравнивание
 * пакетов inodes
 * 0x0100 - задействовано выравнивание
 * области данных
 * 0x0200 - поле shared_vn имеет смысл
 * 0x1000 - включено отслеживание
 * незаписываемых экстентов
 * 0x2000 - используются каталоги версии 2
 */

__uint16_t sb_sectsize; /* размер сектора раздела в байтах */
__uint16_t sb_inodesize; /* размер inode в байтах */
__uint16_t sb_inopblock; /* количество inodes на блок */
char sb_fname[12]; /* имя ФС */
__uint8_t sb_blocklog; /* log2 от sb_blocksize */
__uint8_t sb_sectlog; /* log2 от sb_sectsize */
__uint8_t sb_inodelog; /* log2 от sb_inodesize */
__uint8_t sb_inopblock; /* log2 от sb_inopblock */
__uint8_t sb_agblklog; /* log2 от sb_agblocks */
__uint8_t sb_rextslog; /* log2 от sb_rextents */
__uint8_t sb_inprogress; /* работает mkfs, не монтировать */
__uint8_t sb_imax_pct; /* максимальный процент пространства
 * ФС для inodes */

__uint64_t sb_icount; /* количество выделенных inodes */
__uint64_t sb_ifree; /* количество свободных inodes */
__uint64_t sb_fdblocks; /* количество свободных блоков данных */
__uint64_t sb_frextents; /* количество свободных realtime-экстентов */
xfs_ino_t sb_uquotino; /* inode квот-файла для пользователей */
xfs_ino_t sb_gquotino; /* inodes квот-файла для групп */
__uint16_t sb_qflags; /* флаги квот */
__uint8_t sb_flags; /* флаги */
__uint8_t sb_shared_vn; /* индикатор разделяемого номера версии */
xfs_extlen_t sb_inoalignmt; /* величина выравнивание пакета
 * inodes в блоках */

__uint32_t sb_unit;
__uint32_t sb_width;
__uint8_t sb_dirblklog;
__uint8_t sb_logsectlog; /* log2 от размера сектора журнала */
__uint16_t sb_logsectsize; /* размер сектора для журна в байтах (если
 * он вынесен на отдельный раздел */

__uint32_t sb_logsunit;
__uint32_t sb_features2;
} xfs_sb_t;

```

AGF-блок

AGF-блок содержит служебную информацию, необходимую подсистеме выделения блоков (block allocator). Его структура такова (xfs_ag.h):

```

typedef struct xfs_agf {
__be32      agf_magicnum;      /* magic-номер = XFS_AGF_MAGIC ("XAGF") */
__be32      agf_versionnum;    /* версия заголовка ==
                               * XFS_AGF_VERSION (1) */

__be32      agf_seqno;         /* sequence number starting from 0 */
__be32      agf_length;        /* размер AG в блоках. Все AG, кроме
                               * последней, имеют одинаковый размер,
                               * определенный в поле agblocks
                               * суперблока */

/* Номера корневых блоков для двух деревьев свободных
 * экстенгов - с сортировкой по номеру стартового
 * блока и по длине */
__be32      agf_roots[XFS_BTNUM_AGF];
__be32      agf_spare0;        /* зарезервировано */

/* Количества уровней для двух деревьев свободных
 * экстенгов - с сортировкой по номеру стартового
 * блока и по длине */
__be32      agf_levels[XFS_BTNUM_AGF];
__be32      agf_spare1;        /* зарезервировано */

__be32      agf_flfirst;       /* индекс первого активного элемента
                               * в AGFL-блоке */
__be32      agf_fllast;        /* индекс последнего активного элемента
                               * в AGFL-блоке */
__be32      agf_flcount;       /* количество активных элементов
                               * AGFL-блока */
__be32      agf_freeblks;      /* всего свободных блоков */
__be32      agf_longest;       /* размер самого большого
                               * свободного экстенга */
} xfs_agf_t;

```

Для учета свободных блоков XFS использует не традиционные для многих ФС (например, ext2/3/4, reiserfs, reiser4) битовые карты, а списки свободных экстенгов. Эти массивы проиндексированы в двух В+ деревьях, одно из которых отсортировано по стартовому блоку экстенга, другое – по его длине. Двойное индексирование позволяет не только гибко и быстро находить необходимое количество свободного пространства, но и использовать перспективные политики выделения блоков, такие, как отложенное размещение (delayed allocation).

Алгоритм delayed allocation использует “ленивые” техники назначения физических блоков файлу. Вместо того, что бы выделять блоки файлу в момент его записи в кэш, XFS просто резервирует место в файловой системе, размещая данные в специальных виртуальных экстенгах. Только когда буферизованные данные сбрасываются на диск, виртуальным экстенгам назначаются конкретные блоки. Решение о размещении файла на диске откладывается до момента, когда ФС будет располагать более точной информацией о конечном размере файла. Когда весь файл содержится в памяти, то он обычно может быть размещен в одном куске непрерывного дискового пространства. Файлам, не умещающимся в памяти, алгоритм delayed allocation позволяет быть размещенными гораздо более непрерывно, чем это было бы возможно без его применения.

Механизм отложенного размещения хорошо соответствует концепции современной файловой системы, так как его эффективность возрастает с увеличением объема системной RAM – чем больше данных будет буферизовано в памяти, тем более оптимальные решения по их размещению будет принимать XFS. Кроме того, файлы с малым временем жизни могут вовсе не получить физического воплощения на диске –

XFS просто не успеет принять решение о размещении до их удаления. Такие короткоживущие файлы – обычное дело в UNIX-системах, и механизм *delayed allocation* позволяет существенно уменьшить количество модификаций метаданных, вызванных созданием и удалением таких файлов, а также устранить их влияние на фрагментацию ФС.

Другой плюс отложенного размещения состоит в том, что файлы, записанные беспорядочно, но не имеющие “дыр”, чаще всего будут размещаться на диске рядом. Если все “грязные” данные могут быть буферизованы в памяти, то пространство для этих данных скорее всего будет размещено непрерывно в тот момент, когда они сбрасываются (*flushed*) на диск. Это особенно важно для приложений, пишущих данные в отображенные (*mapped*) файлы, когда случайный доступ – правило, а не исключение.

AGI-блок

AGI-блок хранит служебные данные для аллокатора *inodes*. Его структура такова (*xfs_ag.h*):

```
typedef struct xfs_agi {
    __be32    agf_magicnum;    /* magic-номер == XFS_AGI_MAGIC
    * ("XAGI") */
    __be32    agf_versionnum; /* версия заголовка
    * XFS_AGI_VERSION (1) */
    __be32    agf_seqno;      /* sequence number starting from 0 */
    __be32    agf_length;     /* размер AG в блоках */

    __be32    agi_count;      /* количество выделенных inodes */
    __be32    agi_root;       /* корень дерева inodes */
    __be32    agi_level;      /* высота дерева inodes */
    __be32    agi_freecount;  /* количество свободных inodes */
    __be32    agi_newino;     /* номер последнего выделенного inode */
    __be32    agi_dirino;     /* не используется */
    /*
    * Хэш-таблица inodes, которые были разлинкованы, но
    * все еще достижимы через ссылки в различных списках
    */
    __be32    agi_unlinked[XFS_AGI_UNLINKED_BUCKETS];
} xfs_agi_t;
```

XFS выделяет *inodes* динамически и располагает их на диске в пакетах по 64 штуки. В каждой AG существует B+дерево (на его корень указывает поле *agi_root* структуры *xfs_agi*), в листьях которого хранятся структуры такого вида (*xfs_ialloc_btree.h*):

```
typedef struct xfs_inobt_rec {
    __be32    ir_startino; /* номер первого inode в пакете */
    __be32    ir_freecount; /* количество свободных inodes в пакете */
    __be64    ir_free;     /* битовая карта занятости inodes */
} xfs_inobt_rec_t;
```

Ключем в дереве является номер первого *inode* в пакете, *magic*-номер для всех блоков — "IABT". В XFS номер *inode* вычисляется на основе адреса блока, в котором он хранится. Соответственно, для каждого блока однозначно определен диапазон возможных номеров *inodes*, следовательно, информацию о физическом местоположении пакетов *inodes* хранить не требуется — достаточно номера первого *inode* в пакете.

AGFL-блок

AGFL-блок содержит линейный массив номеров блоков, зарезервированных для использования аллокатором в условиях нехватки свободного места в данной AG. Эти блоки могут быть использованы только для расширения В+ деревьев свободных блоков, и ни для каких других данных.

В только что созданной файловой системе этот список содержит 4 блока — с 4 по 7. По мере фрагментации свободного и заполнения этих блоков XFS заносит в AGFL другие.



Рис. 1. Структура первой AG сразу после создания

Inodes

Каждый inode состоит из четырех частей – ядра, next_unlinked, u и a. Ядро содержит постоянную информацию, характерную для всех типов inodes. Поле di_next_unlinked, отделенное от ядра из-за принятой политики журналирования, по функциям связано с agi_unlinked (см. рис. 2). Вот структура ядра inode (см. xfs_dinode.h):

```
typedef struct xfs_dinode_core {
    __uint16_t    di_magic;           /* inode magic # = XFS_DINODE_MAGIC ("IN") */
    __uint16_t    di_mode;           /* mode и тип файла */
    __int8_t      di_version;        /* версия inode, 1 или 2 */
    __int8_t      di_format;        /* формат потока данных */
    __uint16_t    di_onlink;         /* количество ссылок на inodes версии 1 */
    __uint32_t    di_uid;           /* user id владельца */
    __uint32_t    di_gid;           /* group id владельца */
    __uint32_t    di_nlink;         /* количество ссылок на inode версии 2 */
    __uint16_t    di_projid;        /* project id владельца, только для IRIX */
    __uint8_t     di_pad[8];        /* не используется */
};
```

```

__uint16_t di_flushiter; /* инкрементируется при flusf */
xfs_timestamp_t di_atime; /* время последнего доступа */
xfs_timestamp_t di_mtime; /* время последнего изменения */
xfs_timestamp_t di_ctime; /* время создания/модификации inode */
xfs_fsize_t di_size; /* количество байт в файле */
xfs_drfsbno_t di_nblocks; /* количество использованных
* direct- и btree-блоков */

xfs_extlen_t di_extsize; /* basic/minimum размер экстенда файла */
xfs_extnum_t di_nextents; /* количество экстендов в потоке
* данных */

xfs_aextnum_t di_anextents; /* количество экстендов в потоке
* атрибутов */

__uint8_t di_forkoff; /* смещение потока данных в inode,
* в 64-битных словах от начала u */
__int8_t di_aformat; /* формат потока атрибутов:
* 1 - локальные атрибуты,
* 2 - список экстендов,
* 3 - корень дерева */

__uint32_t di_dmevmask; /* DMIG event mask */
__uint16_t di_dmstate; /* DMIG state info */
__uint16_t di_flags; /* флаги */
__uint32_t di_gen; /* номер поколения */
} xfs_dinode_core_t;

```

Вот формат inode целиком:

```

typedef struct xfs_dinode {
    xfs_dinode_core_t di_core; /* ядро inode */

    xfs_agino_t di_next_unlinked;

    /* Это объединение описывает поток данных (data-fork) */
    union {
        xfs_bmdr_block_t di_bmbt; /* корень дерева */
        xfs_bmbt_rec_32_t di_bmx[1]; /* список экстендов */
        xfs_dir2_sf_t di_dir2sf; /* короткая форма каталога v2 */
        char di_c[1]; /* локальное содержимое */
        xfs_dev_t di_dev; /* устройство */
        uuid_t di_muuid; /* точка монтирования */
        char di_symlink[1]; /* символическая ссылка */
    } di_u;

    /* Это объединение описывает поток атрибутов (attribute-fork) */
    union {
        xfs_bmdr_block_t di_abmbt; /* корень дерева */
        xfs_bmbt_rec_32_t di_abmx[1]; /* список экстендов */
        xfs_attr_shortform_t di_attrsf; /* локальные атрибуты */
    } di_a;
} xfs_dinode_t;

```

Формат di_u определяется типом файла, идентификатор которого хранится в поле di_format ядра inode. Возможные варианты описаны в этом перечислении (xfs_dinode.h):

```

/* Значения для di_format */
typedef enum xfs_dinode_fmt {
    XFS_DINODE_FMT_DEV, /* спец. файл - di_dev */
    XFS_DINODE_FMT_LOCAL, /* локальное содержимое (файл - di_c, короткий
* каталог - di_dir2sf, или symlink - di_symlink) */

    XFS_DINODE_FMT_EXTENTS, /* список экстендов - di_bmx */
    XFS_DINODE_FMT_BTREE, /* корень дерева - di_bmbt */
    XFS_DINODE_FMT_UUID /* не используется */
} xfs_dinode_fmt_t;

```

Если `di_format == XFS_DINODE_FMT_LOCAL`, значит поле "u" хранит внутри себя все содержимое файла. Это могут быть сырые данные файла (`di_c`), текстовая строка с именем цели для символической ссылки (`di_symlink`), или небольшой массив элементов каталога описывающий короткий каталог (`di_dir2sf`, см. далее).

В случае `di_format == XFS_DINODE_FMT_DEV`, "u" интерпретируется как 32-битный дескриптор устройства, содержащий его `minor` и `major` номера — структура `di_dev`.

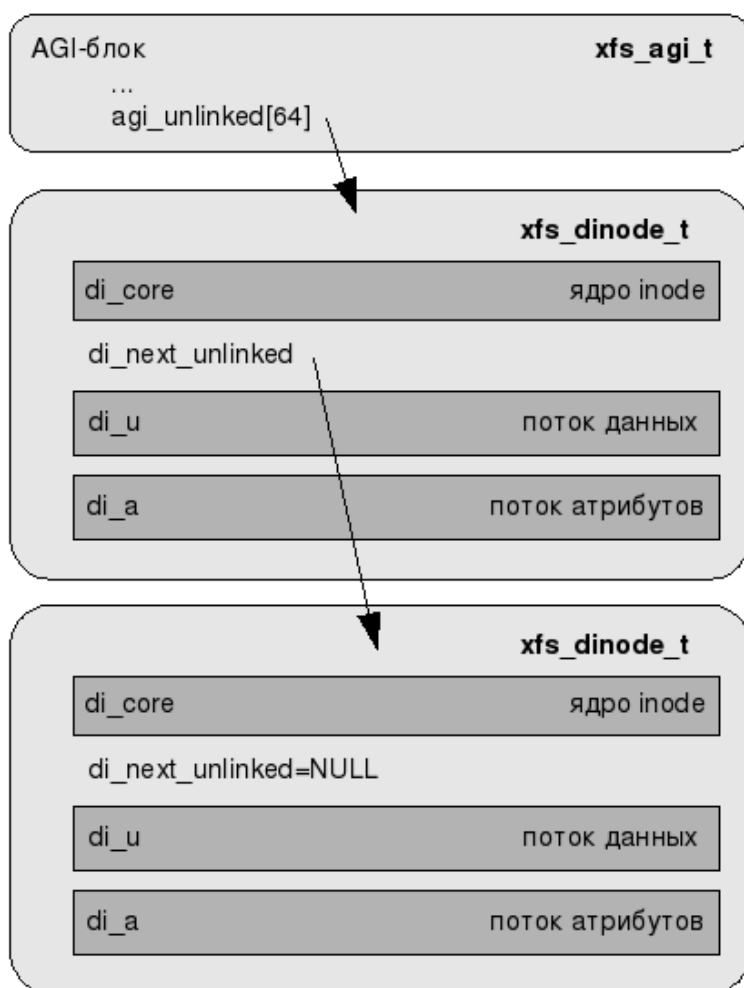


Рис. 2. Список разлинкованных inodes

При `di_format == XFS_DINODE_FMT_EXTENTS`, "u" интерпретируется как массив дескрипторов экстендов (`di_bmx`), содержащих данные файла. Структура дескриптора экстенда такова (`xfs_bmap_btree.h`):

```
typedef struct xfs_bmbt_rec_32
{
    __uint32_t    10, 11, 12, 13;
} xfs_bmbt_rec_32_t;

typedef struct xfs_bmbt_rec_64
{
    __uint64_t    10, 11;
} xfs_bmbt_rec_64_t;
```

В 32-битных ядрах поля структуры интерпретируются следующим образом:

- I0:31 флаг экстента (1 если non-normal)
- I0:0-30 и I1:9-31 это смещение экстента от начала файла (в блоках), служит ключем в дереве выделенных экстентов
- I1:0-8, I2:0-31 и I3:21-31 это номер стартового блока от начала AG, используется в свободных экстентах
- I3:0-20 это количество блоков в экстенте

Для 64-битных ядер:

- I0:63 флаг экстента (1 если non-normal)
- I0:9-62 это смещение экстента от начала файла (в блоках)
- I0:0-8 и I1:21-63 номер стартового блока
- I1:0-20 количество блоков

При `di_format == XFS_DINODE_FMT_BTREE`, "u" считается содержащим корневой узел дерева выделенных экстентов (`di_bmbt`). Каждый внутренний блок этого дерева состоит из трех частей: общего заголовка (см. раздел "B+ деревья" в начале статьи), массива ключей (в этом дереве ключем считается смещение экстента от начала файла) и массива указателей на потомки. Листовые узлы, вместо массива указателей, содержат массив дескрипторов экстентов. `magic`-номер для всех блоков такого дерева — "BMAP".

Атрибуты

Каждый файл в XFS может располагать дополнительным потоком данных — `attribute fork`, описываемым объединением "a" структуры `inode`. Как и в случае с данными, `inode` может содержать поток атрибутов в короткой форме (`di_abmbt`). Структура `di_abmbt` имеет следующий формат (см. `xfs_attr_sf.h`):

```
typedef struct xfs_attr_shortcode {
    struct xfs_attr_sf_hdr {           /* заголовок */
        __be16 totsize;               /* размер списка */
        __u8  count;                  /* количество активных элементов */
    } hdr;
    struct xfs_attr_sf_entry {
        __uint8_t namelen;             /* длина имени атрибута */
        __uint8_t valuelen;           /* длина значения атрибута */
        __uint8_t flags;               /* биты флагов (см. xfs_attr_leaf.h) */
        __uint8_t nameval[1];         /* объединенные строки имени и
                                        * значения атрибута */
    } list[1];                         /* список переменной длины */
} xfs_attr_shortcode_t;
```

Если все атрибуты в `inode` не умещаются, "a" интерпретируется в качестве массива дескрипторов экстентов (`di_abmx`). В случае нехватки места и для списка экстентов, в `di_a` помещается корневой узел дерева атрибутов (`di_abmbt`). Внутренние узлы этого дерева имеют традиционный формат (`magic 0xfbee`), листовые же устроены несколько иначе:

заголовок		массив индексов и хэшей	буфер актуальных данных	
header info	массив freemap		локальные атрибуты	удаленные атрибуты

Структура заголовка листового узла (см. xfs_attr_leaf.h):

```
#define XFS_ATTR_LEAF_MAPSIZE    3    /* доступное количество free-слотов */

/* Формат free-карты */
typedef struct xfs_attr_leaf_map {
    __be16 base;                /* база свободного региона */
    __be16 size;                /* длина свободного региона */
} xfs_attr_leaf_map_t;

/* Заголовок листового блока */
typedef struct xfs_attr_leaf_hdr {
    xfs_da_blkinfo_t info;      /* этот заголовок используется для построения
    * двусвязных списков блоков одного уровня
    * (см. xfs_da_btree.h) */
    __be16 count;               /* количество активных leaf_entry */
    __be16 usedbytes;           /* количество байт в буфере */
    __be16 firstused;           /* первый используемый байт в массиве имен */
    __u8 holes;                 /* != 0 если блок нуждается в упаковке */
    __u8 pad1;
    xfs_attr_leaf_map_t freemap[XFS_ATTR_LEAF_MAPSIZE];
} xfs_attr_leaf_hdr_t;
```

Массив freemap хранит описатели для N (XFS_ATTR_LEAF_MAPSIZE=3 в текущей реализации) наибольших свободных регионов. Если места в узле не достаточно, вызываются процедуры упаковки узла и, если не помогло, — его разбиения.

Структура xfs_da_blkinfo_t определена в xfs_da_btree.h и используется для построения двусвязных списков блоков одного уровня в деревьях именованных объектов — атрибутов и каталогов (xfs_da_btree.h):

```
typedef struct xfs_da_blkinfo {
    __be32 forw;                /* предыдущий блок в списке */
    __be32 back;                /* следующий блок в списке */
    __be16 magic;                /* тип блока */
    __be16 pad;                 /* unused */
} xfs_da_blkinfo_t;

#define XFS_DA_NODE_MAGIC        0xfebe    /* внутренний узел */
#define XFS_ATTR_LEAF_MAGIC      0xfbee    /* лист дерева атрибутов */
#define XFS_DIR2_LEAF1_MAGIC     0xd2f1    /* лист дерева одноблочной
    * директории */
#define XFS_DIR2_LEAFN_MAGIC     0xd2ff    /* лист дерева многоблочной
    * директории */
```

Далее следует массив хэшей/индексов:

```
typedef struct xfs_attr_leaf_entry {
    __be32 hashval;             /* хэш имени атрибута */
    __be16 nameidx;             /* индекс атрибута в буфере данных */
    __u8 flags;                 /* LOCAL/ROOT/SECURE/INCOMPLETE флаги */
    __u8 pad2;                 /* unused pad byte */
} xfs_attr_leaf_entry_t;
```

Листовой блок заканчивается буфером актуальных данных, содержащим имена и значения атрибутов. Значения могут быть локальными (хранятся прямо в буфере) и удаленными (в буфере только указатель на выделенный блок).

```
/* Локальный атрибут */
typedef struct xfs_attr_leaf_name_local {
    __be16 valuelen;          /* длина значения */
    __u8  namelen;           /* длина имени */
    __u8  nameval[1];       /* объединенные строки имени и значения
                             * атрибута */
} xfs_attr_leaf_name_local_t;

/* Удаленный атрибут */
typedef struct xfs_attr_leaf_name_remote {
    __be32 valueblk;         /* номер блока, хранящего значение */
    __be32 valuelen;        /* длина значения */
    __u8  namelen;          /* длина имени */
    __u8  name[1];          /* имя */
} xfs_attr_leaf_name_remote_t;
```

Вот структура листового узла дерева атрибутов в сборе:

```
typedef struct xfs_attr_leafblock {
    xfs_attr_leaf_hdr_t  hdr;          /* заголовок */
    xfs_attr_leaf_entry_t  entries[1]; /* индексы */
    xfs_attr_leaf_name_local_t  namelist; /* локальные атрибуты */
    xfs_attr_leaf_name_remote_t  valuelist; /* удаленные атрибуты */
} xfs_attr_leafblock_t;
```

Директории

Содержимое директории в XFS хранится либо прямо в inode (каталог короткой формы), либо адресуется через дерево блочной карты файла. Все смещения в структурах каталога логические, т.е. для поиска физического положения тех или иных элементов файловая система обращается к карте блоков файла и выполняет соответствующие преобразования. В пределах файла выделены три диапазона:

- 0-0x7fffffff — область блоков данных
- 0x800000000-0xffffffff — область хэш-блоков
- 0x1000000000-0x17ffffff область freeindex-данных

Каталог в XFS может храниться в четырех форматах:

- короткая форма
- одноблочный каталог
- несколько data-блоков, единственный freeindex-блок
- блоки данных с B+ деревом, несколько freeindex-блоков

Каталог короткой формы

Рассмотрим в каком виде XFS хранит директории. Прямо в inode может содержаться каталог короткой формы, описанный в `xfs_dir2_sf.h`:

```

/* Формат номера inode */
typedef union {
    xfs_dir2_ino8_t    i8;
    xfs_dir2_ino4_t    i4;
} xfs_dir2_inou_t;

/* Заголовок каталога короткой формы */
typedef struct xfs_dir2_sf_hdr {
    __uint8_t          count;           /* количество элементов */
    __uint8_t          i8count;        /* количество 64-битных номеров inodes */
    xfs_dir2_inou_t    parent;        /* номер inode родительского каталога */
} xfs_dir2_sf_hdr_t;

/* Элемент каталога короткой формы */
typedef struct xfs_dir2_sf_entry {
    __uint8_t          namelen;        /* длина имени */
    xfs_dir2_sf_off_t  offset;        /* saved offset */
    __uint8_t          name[1];       /* имя */
    xfs_dir2_inou_t    inumber;       /* номер inode для данного имени */
} xfs_dir2_sf_entry_t;

/* Каталог короткой формы */
typedef struct xfs_dir2_sf {
    xfs_dir2_sf_hdr_t  hdr;           /* заголовок */
    xfs_dir2_sf_entry_t list[1];     /* массив элементов */
} xfs_dir2_sf_t;

```

Одноблочный каталог

Если каталог не помещается в inode, XFS выделяет под него отдельный блок, размещая там структуры т.н. одноблочной директории (xfs_dir2_block.h):

```

/* Одноблочный каталог */
typedef struct xfs_dir2_block {
    xfs_dir2_data_hdr_t  hdr;         /* заголовок */
    xfs_dir2_data_union_t u[1];      /* массив элементов каталога */
    xfs_dir2_leaf_entry_t leaf[1];   /* массив индексов */
    xfs_dir2_block_tail_t tail;      /* хвост блока */
} xfs_dir2_block_t;

```

Заголовок этого блока имеет следующий формат (xfs_dir2_data.h):

```

/* XFS_DIR2_DATA_FD_COUNT равно 3 */
typedef struct xfs_dir2_data_hdr {
    __be32          magic;
    xfs_dir2_data_free_t bestfree[XFS_DIR2_DATA_FD_COUNT];
} xfs_dir2_data_hdr_t;

```

Элемент каталога устроен так (xfs_dir2_data.h):

```

/* Активный элемент каталога */
typedef struct xfs_dir2_data_entry {
    __be64          inumber;          /* номер inode */
    __u8           namelen;          /* длина имени */
    __u8           name[1];         /* имя */
    __be16         tag;              /* смещение этого элемента в блоке */
} xfs_dir2_data_entry_t;

/* Свободный элемент каталога */

```

```

typedef struct xfs_dir2_data_unused {
    __be16      freetag;      /* XFS_DIR2_DATA_FREE_TAG */
    __be16      length;      /* длина свободного участка */
    __be16      tag;         /* смещение этого элемента в блоке */
} xfs_dir2_data_unused_t;

/* Обобщенный элемент каталога */
typedef union {
    xfs_dir2_data_entry_t    entry;
    xfs_dir2_data_unused_t  unused;
} xfs_dir2_data_union_t;

```

Индексная информация к массиву элементов каталога хранится в списке структур такого формата (xfs_dir2_leaf.h):

```

typedef struct xfs_dir2_leaf_entry {
    __be32      hashval;     /* хэш имени */
    __be32      address;     /* адрес элемента */
} xfs_dir2_leaf_entry_t;

```

Наконец, так устроен хвост блока (xfs_dir2_block.h):

```

typedef struct xfs_dir2_block_tail {
    __be32      count;       /* количество листовых элементов */
    __be32      stale;       /* количество устаревших листовых элементов */
} xfs_dir2_block_tail_t;

```

Многоблочный каталог

Когда каталог перестает уместиться в одном блоке, XFS разворачивает под его хранение древовидную структуру, состоящую из data-блоков, хранящих собственно элементы каталога — т.е. имена файлов и номера их inodes, и т.н. leaf-блоков, содержащих индексную информацию по data-блокам — хэши имен и смещения соответствующих элементов в пределах каталога. К этим структурам привязываются так называемые freeindex-блоки, с помощью которых отслеживаются свободные участки в файле, содержащем каталог, в больших каталогах над этой информацией так же может надстраиваться древовидная структура. Логическое пространство в пределах файла каталога считается 8-байтными словами — на это рассчитаны все структуры учета свободных участков.

Итак, блоки данных (xfs_dir2_data.h):

```

/* Активный элемент каталога */
typedef struct xfs_dir2_data_entry {
    __be64      inumber;     /* номер inode */
    __u8        namelen;     /* длина имени */
    __u8        name[1];    /* имя */
    __be16      tag;         /* смещение этого элемента в блоке */
} xfs_dir2_data_entry_t;

/* Свободный элемент каталога */
typedef struct xfs_dir2_data_unused {
    __be16      freetag;     /* XFS_DIR2_DATA_FREE_TAG */
    __be16      length;     /* длина свободного участка */
    __be16      tag;         /* смещение этого элемента в блоке */
} xfs_dir2_data_unused_t;

```

```

/* Обобщенный элемент каталога */
typedef union {
    xfs_dir2_data_entry_t    entry;
    xfs_dir2_data_unused_t  unused;
} xfs_dir2_data_union_t;

```

Структура хэш-блоков (т.н. листовые блоки, `xfs_dir2_leaf.h`):

```

/* Заголовок листового блока */
typedef struct xfs_dir2_leaf_hdr {
    xfs_da_blkinfo_t    info;          /* DA-заголовок, см. xfs_da_btree.h */
    __be16              count;        /* количество элементов */
    __be16              stale;
} xfs_dir2_leaf_hdr_t;

/* Элемент листового узла */
typedef struct xfs_dir2_leaf_entry {
    __be32              hashval;      /* хэш имени */
    __be32              address;      /* смещение в пределах файла каталога */
} xfs_dir2_leaf_entry_t;

/* Хвост */
typedef struct xfs_dir2_leaf_tail {
    __be32              bestcount;
} xfs_dir2_leaf_tail_t;

/* Листовой блок в сборе */
typedef struct xfs_dir2_leaf {
    xfs_dir2_leaf_hdr_t    hdr;          /* заголовок */
    xfs_dir2_leaf_entry_t  ents[1];     /* элементы */
    xfs_dir2_data_off_t    bests[1];    /* список свободных участков в
                                         * диапазоне листовых блоков */
    xfs_dir2_leaf_tail_t   tail;        /* хвост */
} xfs_dir2_leaf_t;

```

Не трудно понять, что древовидными эти структуры можно назвать лишь с натяжкой — в сущности, каталоги XFS индексируются с помощью хэширования, при чем даже не расширенного. Управление свободным пространством в каталоге так же не на высоте. Все это приводит к отставанию XFS при работе с большими каталогами от таких продвинутых в этом отношении файловых систем, как `reiserfs` и, в особенности, `reiser4`.

Квоты

Информация о квотах хранится в файлах, адресуемых полями `ino` (квоты для пользователей) и `g` (квоты для групп) суперблока. Каждый блок квот-файла содержит постоянное количество квот-записей. Квот-запись в настоящее время имеет размер 136 байт, поэтому ФС с 4-Кб блоком хранит 30 записей на блок. В файле квот-записи индексируются по идентификатору. Формат квот-записи (`xfs_disk_dquot_t`) определен в `xfs_quota.h`:

- `magic` ("DQ")
- `version`
- `флаг`
- `id` – идентификатор (пользователя/группы)
- `blk_hardlimit` – абсолютное ограничение на количество блоков
- `blk_softlimit` – предпочитаемое ограничение на количество блоков
- `ino_hardlimit` – абсолютное ограничение на количество `inodes`

- `ino_softlimit` – предпочитаемое ограничение на количество `inodes`
- `bcount` – количество уже использованных блоков
- `icount` – количество уже использованных `inodes`
- `btimer` – время, когда обслуживание пользователя будет прекращено, если предпочитаемое ограничение по блокам уже достигнуто
- `itimer` – время, когда обслуживание пользователя будет прекращено, если предпочитаемое ограничение по `inodes` уже достигнуто
- `bwarns` – количество предупреждений, выдаваемых при достижении предпочитаемого ограничения по блокам
- `iwarns` – количество предупреждений, выдаваемых при достижении предпочитаемого ограничения по `inodes`
- `rtb_hardlimit` – абсолютное ограничение на количество RT-блоков
- `rtb_softlimit` – предпочитаемое ограничение на количество RT-блоков
- `rtbtimer` – время, когда обслуживание пользователя будет прекращено, если предпочитаемое ограничение по RT-блокам уже достигнуто
- `rtbwarns` – количество предупреждений, выдаваемых при достижении предпочитаемого ограничения по RT-блокам

Подсистема real-time ввода-вывода

XFS может выделять на диске специальную секцию реального времени, к которой применяются упрощенные механизмы доступа и выделения блоков и `inodes` (в том числе битовые карты), позволяющие стабилизировать время выполнения дисковых операций. Ядро IRIX содержит особые методы в менеджерах томов и памяти, которые обеспечивают поддержку более производительного `realtim`-доступа, в Linux же таких возможностей нет, поэтому секция реального времени обрабатывается здесь не быстрее секции данных. Основная идея — постоянство времени выполнения операций.

Если файловая система имеет `realtime`-подраздел, поле `rbmino` суперблока ссылается на файл, содержащий битовую карту области реального времени. Каждый бит карты контролирует выделение простого `realtime`-экстента. Карта обрабатывается 32-битными словами. Поле `atime inode`, описывающего эту карту, содержит счетчик, используемый для хранения указателя на первый свободный блок.

Поле `rbsumino` суперблока ссылается на файл, содержащий двумерный массив 16-битных значений, с помощью которого несколько ускоряется поиск необходимого экстента по битовой карте `realtime`-блоков. В первом измерении массив упорядочен по размеру (представлены 16 диапазонов возможного размера `realtime`-экстентов), во втором – по смещению экстентов с данным размером в битовой карте. Это позволяет быстро отыскать свободный `realtime`-экстент нужного размера.

Журналирование

Задачей менеджера журнала является обеспечение сервиса, позволяющего быстро и надежно восстанавливаться файловую систему после сбоя. Это также увеличивает производительность для некоторых операций, типа обновления метаданных. Благодаря использованию этого механизма другие сервисы журналируют изменения в метаданных ФС (в `inodes`, каталогах, пуле свободного пространства), а менеджер журнала группирует множество запросов к диску в один большой синхронный запрос, выполняемый аппаратурой гораздо быстрее. Однажды зарегистрированные,

клиенты log-менеджера больше не нуждаются в немедленном сбросе своих грязных данных, эту работу выполняет log-менеджер, попутно группируя запросы на запись и обеспечивая целостность файловой системы в случае сбоя. Изменения в пользовательских данных не журналируются, т.к. их сохранность инвариантна к целостности ФС.

Журнал XFS разбит на 2 части – дисковую и рабочую (in-core). Назначенные на журналирование операции сначала попадают в рабочий лог, который представляет из себя FIFO-очередь, удерживающую транзакции в памяти до тех пор, пока некоторое событие не вызовет их принудительный сброс в дисковый лог. Операции, хранимые в рабочем журнале, называются активными.

Дисковый журнал представляет собой непрерывную последовательность дисковых блоков, обрабатываемых отдельно от блоков данных XFS, и является, по сути, замкнутой очередью. Операции, хранящиеся в дисковом логе, называются зафиксированными.

Набор связанных операций, которые должны быть атомарно применены к диску, называется транзакцией.

Журнальные записи

Для оптимизации ввода/вывода активные операции группируются вместе в log-записи и только потом направляются на диск. Существуют 3 события, которые могут вызвать фиксацию рабочей очереди:

- Переполнение рабочего журнала
- Поступление в log-менеджер запроса на немедленный сброс операции на диск
- Окончание некоторого таймаута

Log-записи состоят из двух частей – заголовка и тела. Заголовок содержит некую общую информацию обо всей log-записи, тогда как тело хранит связанные log-операции. Заголовок log-записи состоит из:

- номер записи
- номер последней сброшенной записи
- длина тела записи в 64-битных словах
- magic-номер
- Количество log-операций в записи

Log-операции содержатся в теле записи также разбитыми на 2 части – заголовок и тело операции. Тело операции – это поток выровненных по 4-байтной границе снапшотов, отражающих состояние метаданных после внесения изменений. Заметим, что это подразумевает временное упорядочивание записей. Метаданные могут быть сброшены на диск только после того, как соответствующие log-записи внесены в дисковый лог. Заголовок log-записи содержит:

- идентификатор инициатора операции
- длина тела в байтах
- ID транзакции

- тип операции

Номер записи представляет собой 64-битное число; младшие 32 бита содержат номер соответствующего блока в журнале, старшие — инкрементальный счетчик.

Дисковая структура

Каждая log-запись предваряется информационным заголовком (размером не более сектора), который содержит:

```
typedef struct xlog_rec_header {
    uint    h_magicno;        /* ID log-записи */
    uint    h_cycle;         /* номер цикла записи */
    int     h_version;       /* версия log-записи */
    int     h_len;           /* длина записи в байтах */
    xfs_lsn_t h_lsn;         /* номер сектора для данной записи */
    xfs_lsn_t h_tail_lsn;    /* номер сектора первой не сброшенной записи */
    uint    h_chksm;        /* контрольная сумма тела записи,
                           * если не используется - 0 */
    int     h_prev_block;    /* номер блока предыдущей LR */
    int     h_num_logops;    /* количество log-операций в этой LR */

    uint    h_cycle_data[XLOG_HEADER_CYCLE_SIZE / BBSIZE];

    int     h_fmt;           /* формат log-записи */
    uuid_t  h_fs_uuid;       /* UUID файловой системы */
    int     h_size;          /* размер */
} xlog_rec_header_t;
```

Так как восстановление после сбоя должно быть очень быстрым, оперативный поиск начала дискового журнала является приоритетной задачей log-менеджера. Последняя актуальная log-запись будет удовлетворять следующим условиям:

- после нее нет действительных log-записей
- контрольная сумма ее тела будет соответствовать той, что хранится в заголовке

Log-менеджер XFS ищет начало журнала, выполняя случайные выборки по всему логу и методом последовательных приближений отыскивая log-запись с наибольшим номером. Затем около нее ищется последняя запись с верной контрольной суммой, из заголовка которой извлекается номер первой не сброшенной записи (`h_tail_lsn`) — она и считается началом журнала.

Мнение

XFS довольно сложная — как по идеям, так по реализации — файловая система. За счет эффективных, хорошо продуманных алгоритмов распределения дискового пространства, отложенного размещения и хорошо распараллеленной обработки пользовательских запросов показывает хорошие результаты на обработке больших и средних файлов. Излишне сложная и недостаточно оптимизированная схема обработки каталогов не позволяет ей опережать такие ФС, как `reiserfs`, `reiser4` и `btrfs` на больших директориях (`ext2/3/4` здесь отстают от нее довольно существенно, `JFS` — примерно на одном уровне) — и это, пожалуй, ее единственная слабость.

Источники

1. Исходники XFS ядра Linux-2.6.27
2. man 5 xfs, man 8 xfs_db
3. XFS design documents, oss.sgi.com/projects/xfs/design_docs/
4. "Масштабируемость в XFS", filesystems.nm.ru/my/xfs_arch.pdf
5. "XFS filesystem structure",
oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf