

author: Пешеходов Андрей (filesystems@nm.ru)  
released: 18.05.2010  
modified: 18.05.2010

Статья была опубликована в журнале "Системный администратор", № 1-2 (январь-февраль) 2010 года.

## Не новое, но хорошо доработанное старое: взгляд на ext4

Ext3 много лет является наиболее популярной файловой системой для Linux. Для соответствия возможностям новых жестких дисков и требованиям времени на ее основе была разработана ФС нового поколения – ext4, сочетающая в себе улучшенные масштабируемость и производительность при поддержке больших файловых систем с сохранением надежности и стабильности ext3. Ext4 подходит для самых разнообразных рабочих нагрузок и способна полностью заменить ext3 в качестве "файловой системы Linux".

Здесь мы рассмотрим мотивы разработки ext4, ее новые возможности, методы миграции с ext3 и сравним ее с другими файловыми системами.

### Введение

Ext3 стала столь популярной файловой системой благодаря своей надежности, богатому набору возможностей, относительно хорошей производительности и отличной совместимости с предыдущими версиями Extended File System. Консервативный дизайн ext3 создал ей репутацию высоконадежной, но несколько ограниченной – в работе с большими хранилищами – файловой системы.

Одно из наиболее неприятных ограничений ext3 – максимальный размер файловой системы в 16 Tb. Многие корпоративные решения, в наше время, столкнулись с этим лимитом, более того, многотерабайтные массивы становятся все более доступны и домашним пользователям.

Для снятия этого ограничения в августе 2006-го была опубликована серия патчей, дающих ext3 две ключевые возможности: поддержку больших разделов и учет блоков на экстендах (необходимый из-за того, что обработка битовых карт даже на терабайтном разделе довольно сильно нагружает CPU и память). Эти патчи необратимо меняли дисковый формат и нарушали обратную совместимость. Из соображений поддержания стабильности кода ext3 разработчики решили создавать на основе этих патчей новую файловую систему, назвав ее ext4.

Основная цель новой ФС – решить проблемы производительности, надежности и масштабируемости, характерные для ext3. На вопрос об использовании XFS или разработки новой ФС с нуля есть простой ответ – необходимо было обеспечить огромному количеству пользователей ext3 возможность легко обновить их файловые системы – как это было сделано при переходе с ext2. Также разработчикам не хотелось пренебрегать значительными усилиями, вложенными в надежность и функциональность ext3 и e2fsck, а

сосредоточится на достаточно быстром добавлении новых возможностей.

Так была рождена ext4, присутствующая в основной ветке начиная с ядра 2.6.19 и помеченная стабильной с 2.6.28 (после двух лет разработки).

## **Масштабирование**

16-терабайтный лимит размера раздела в ext3 обусловлен 32-битным номером блока. Логичным решением проблемы является использование большего количества битов под этот параметр по всему коду ФС.

В экстен-патче для ext3 вводились 48-битные номера блоков, сохраненные и в ext4. Теперь поддерживаются разделы размером до 1 Eb при 4-килобайтном блоке.

После расширения номера блока до 48 бит потребовалось внести изменения в метаданные ext4 – суперблок, дескриптор группы блоков и журнал. Новые поля были добавлены в конец суперблока, расширяя s\_free\_blocks\_count (количество свободных блоков), s\_blocks\_count () и s\_r\_blocks\_count до 64 бит. Аналогичным образом был скорректирован и дескриптор группы блоков – были расширены указатели на битовую карту и таблицу inodes.

Уровень журналирования блоков, JBD, был переработан в JBD2 для поддержки новых возможностей ext4. Хотя в настоящий момент подсистему JBD2 использует только ext4, потенциально она может обеспечить журналирование для любой 32-х или 64-битной файловой системы.

48-битная разрядность номера блоков была выбрана потому, что, во-первых, предела размера ФС в 1 Eb должно хватить на много лет вперед, и, во-вторых, полная проверка с помощью e2fsck экзабайтного раздела займет 119 лет при современных скоростях дисков.

Несмотря на расширение номера блока до 48 бит, размер раздела ext4 все еще ограничен количеством групп блоков в файловой системе. В ext3, из соображений безопасности, все дескрипторы групп блоков продублированы в первой из них. Т.к. новый дескриптор группы блоков имеет длину 64 байта, ext4 могла бы иметь объем не более 256 Tb.

Решение проблемы – использовать метагруппы блоков (META\_BG), существующие в ext3 начиная с ядра 2.6.0. С опцией META\_BG ext4 разбивается на несколько метагрупп, каждая из которых является кластером такого количества групп блоков, дескрипторы которых могут храниться в одном блоке ФС. В 4-килобайтном блоке помещается 64 дескриптора нового формата, т.е. одна метагруппа может адресовать до 8 Gb дискового пространства. Использование метагрупп позволяет вынести дескрипторы конкретных групп блоков из перегруженной первой группы раздела в первые группы каждой метагруппы. Резервные копии дескрипторов хранятся во второй и последней группе блоков каждой метагруппы. Все эти меры позволяют адресовать с помощью групп блоков 1 Eb дискового пространства.

## Экстенты

Файловая система ext3 отслеживает блоки данных файлов и каталогов с помощью косвенно-блочной схемы. Этот подход достаточно эффективен для сильно фрагментированных или маленьких файлов, но очень накладен для больших хорошо упакованных файлов – особенно на операциях удаления/усечения.

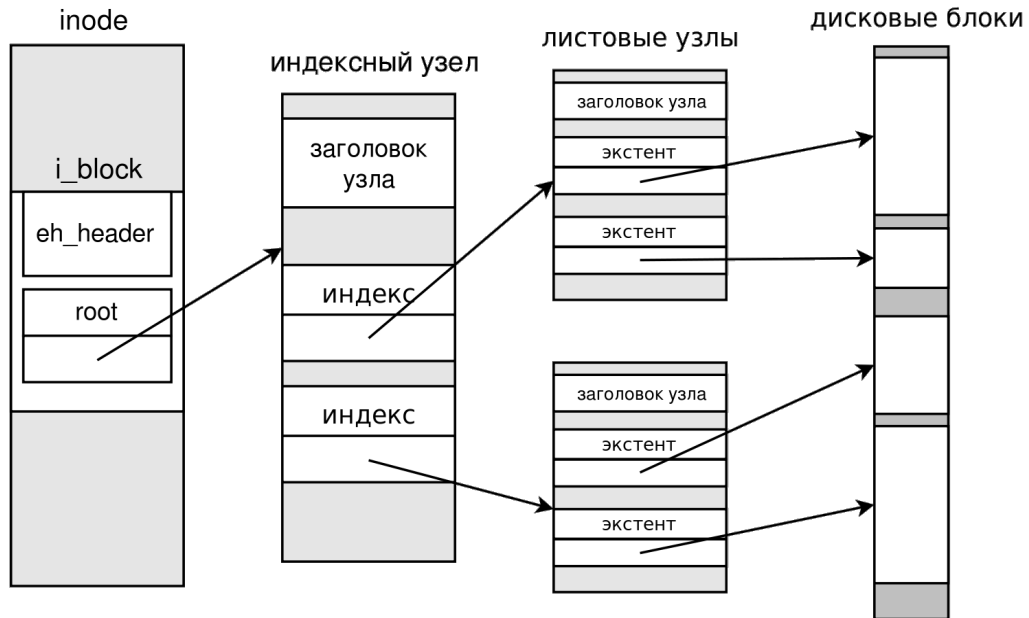


Рис. 1. Дерево экстенгов ext4

Экстентом называется дескриптор, определяющий участок в несколько непрерывно расположенных дисковых блоках. В ext4 экстент может адресовать до 128 Mb дискового пространства при 4-килобайтном блоке. 4 экстенга могут храниться прямо в inode – их вполне достаточно для небольших или нефрагментированных файлов. Когда встроенных экстенгов не хватает, для адресации блоков файла используется дерево экстенгов постоянной глубины (см. Рис. 1). Корень этого дерева хранится в inode, сами экстенга – в листьях. Каждый узел дерева начинается с заголовка (fs/ext4/ext4\_extents.h):

```
/*
 * Каждый узел (листовой, индексный или корневой), начинается с этого заголовка
 */
struct ext4_extent_header {
    __le16 eh_magic;          /* в будущем возможны несколько форматов узлов,
                             * тогда пригодится этот magic */
    __le16 eh_entries;       /* количество элементов */
    __le16 eh_max;           /* максимально элементов */
    __le16 eh_depth;        /* глубина дерева, начиная с этого уровня */
    __le32 eh_generation;    /* номер поколения дерева */
};

/*
 * Структура индекса в дереве экстенгов
 */
struct ext4_extent_idx {
```

```

    __le32 ei_block; /* индекс покрывает логические блоки, начиная с этого */
    __le32 ei_leaf_lo; /* указатель на блок следующего уровня (младшие 32 бита) */
    __le16 ei_leaf_hi; /* (старшие 16 бит) */
    __u16 ei_unused;
};

/*
 * Экстент ext4
 */
struct ext4_extent {
    __le32 ee_block; /* логическое начало экстента */
    __le16 ee_len; /* длина экстента в блоках */
    __le16 ee_start_hi; /* физическое начало экстента (старшие 16 бит) */
    __le32 ee_start_lo; /* (младшие 32 бита) */
};

```

В будущем, специально для эффективного хранения фрагментированных файлов, возможно введение нового типа экстентов (с отличным magic-идентификатором), адресующих блоки по косвенной схеме. Также, для повышения надежности определения разрушений, не исключено дополнение узлов дерева экстентов, помимо заголовка, еще и "хвостом", в котором будут содержаться номер inode, номер поколения и контрольная сумма.

### Большие файлы

В файловой системе ext3 размер файла ограничен 32-битным полем `i_blocks` в `inode`, содержащем количество секторов (512 байт), занимаемых файлом. В итоге, максимальный размер файла в ext3 ограничен величиной  $2^{32} * 512 = 2\text{Tb}$ , что для ФС нового поколения недостаточно.

В ext4 эта проблема решена достаточно прямолинейно и бесхитростно: разрядность поля `i_blocks` увеличена до 48 бит, размер теперь исчисляется в блоках. В итоге размер файла ext4 ограничен только 32-битным номером логического блока в текущем формате экстента и составляет 16 Tb. В будущем планируется снятие и этого ограничения и достижение полной 48-битной емкости в несколько Eb.

### Большие каталоги

Некоторые приложения уже в наше время оперируют миллиардами файлов, и даже замахиваются на триллионы. Теоретически ext4, располагая 32-битным номером `inode`, может оперировать миллиардами файлов, однако на практике эта величина не достижима из-за архаичного статического выделения `inodes`, пришедшего в Extended Filesystem еще из BSD FFS. Т.е. количество файлов на разделе жестко задается во время создания ФС. Задумавшись о реализации динамического выделения `inodes` с увеличенными 64-битными номерами, разработчики остановились на следующих моментах:

- **производительность:** необходим эффективный способ отображения номеров `inodes` в их физическое положение на диске
- **устойчивость к ошибкам оборудования:** `e2fsck` должна иметь возможность быстро отыскивать все `inodes` после разрушения ФС

- **совместимость:** необходимо решать проблемы обработки 64-битных номеров inodes на 32-битных машинах

При динамическом выделении inodes они больше не обладают фиксированным положением на диске. Одним из способов отображения номера inode на номер содержащего его блока является прямое кодирование координат inode в его номере, как это сделано в XFS. Недостатки этого метода проявляются при усечении или дефрагментации файловой системы – то есть в тех случаях, когда inodes требуется перемещать. В этом случае сервисной программе придется вникать и вносить изменения в структуру каталогов, что ведет к существенной потере производительности и увеличивает риск разрушения ФС в случае сбоя оборудования во время работы программы. Панацеей здесь может стать введение дополнительных карт, отображающих старые координаты inodes на новые, что, тем не менее, трудно назвать хорошей идеей. Решение же проблемы поиска динамически выделяемых inodes с помощью индексированных их бинарных деревьев (как это сделано в подавляющем большинстве файловых систем) приведет к полной переделке структуры ФС, утраты аккумулялирующей роли групп блоков и, фактически, выльется в создание полностью новой и ни с чем не совместимой ФС, что неприемлемо. То же можно сказать и о расширении номера inode до 64 бит.

Выгоды от динамического выделения и 64-битных номеров inodes очевидны, однако объем требуемых изменений дисковой структуры необычайно велик, поэтому в настоящее время эти вопросы все еще находятся в стадии обсуждения. Соответствующий функционал не вошел в релиз, и, видимо, будет реализован не скоро (а, по мнению автора, не будет реализован вообще – в свете успехов в разработке btrfs ([btrfs.wiki.kernel.org](http://btrfs.wiki.kernel.org)) и все большего внимания к ней со стороны не только пользователей, но и многих мэйнтейнеров ядра).

### **Масштабируемость каталогов**

Объем каталога ext3 ограничен величиной в 32 000 файлов. В ext4 этот лимит полностью устранен – то есть количество файлов в одном каталоге там не ограничено. Для поддержки больших каталогов в ext4, вместо односвязного списка, используется схема индексирования их элементов с помощью так называемых NTTree-структур – своего рода B-деревьев постоянной глубины, индексированных элементов директории по 32-битному хэшу имени. Для каталогов, содержащих более 10 000 файлов, производительность поиска имени возросла в 50-100 раз относительно ext3.

### **Inodes и расширенные атрибуты**

Ext3 поддерживает inodes различного размера, задаваемого во время mkfs параметром `-l [inodes_size]`. В ext4 минимальный размер inode увеличен вдвое – до 256 байт. Для сохранения совместимости с кодом драйвера ext3 и e2fsck размеченная часть inode имеет старый формат. В дополнительной секции расположены несколько новых полей (к примеру, наносекундные временные штампы) и динамическая область, используемая под расширенные атрибуты (extended attributes, EAs).

```

/* Дискový inode ext4s */
struct ext4_inode {
    __le16 i_mode;           /* флаги доступа и типа */
    __le16 i_uid;           /* с */
    __le32 i_size_lo;       /* размер в байтах */
    __le32 i_atime;         /* время доступа */
    __le32 i_ctime;         /* время модификации inode */
    __le32 i_mtime;         /* время модификации данных файла */
    __le32 i_dtime;         /* время удаления */
    __le16 i_gid;           /* младшие 16 бит GID владельца */
    __le16 i_links_count;   /* количество ссылок на inode */
    __le32 i_blocks_lo;     /* количество блоков (младшие биты) */
    __le32 i_flags;         /* флаги */
    union {
        struct {
            __le32 l_i_version; /* младшие 32 бита версии */
        } linux1;
        struct { /* не Linux */
            __u32 h_i_translator;
        } hurd1;
        struct { /* не Linux */
            __u32 m_i_reserved1;
        } masix1;
    } osd1;
    __le32 i_block[EXT4_N_BLOCKS]; /* указатели на блоки данных файла */
    __le32 i_generation; /* номер поколения (для NFS) */
    __le32 i_file_acl_lo; /* указатель на ACL (младшие биты) */
    __le32 i_size_high;
    __le32 i_obso_faddr; /* не используется */
    union {
        struct {
            __le16 l_i_blocks_high; /* количество блоков (старшие биты) */
            __le16 l_i_file_acl_high; /* указатель на ACL (старшие биты) */
            __le16 l_i_uid_high; /* старшие 16 бит UID владельца */
            __le16 l_i_gid_high; /* старшие 16 бит GID владельца */
            __u32 l_i_reserved2;
        } linux2;
        struct { /* не Linux */
            __le16 h_i_reserved1;
            __u16 h_i_mode_high;
            __u16 h_i_uid_high;
            __u16 h_i_gid_high;
            __u32 h_i_author;
        } hurd2;
        struct { /* не Linux */
            __le16 h_i_reserved1;
            __le16 m_i_file_acl_high;
            __u32 m_i_reserved2[2];
        } masix2;
    } osd2;
    __le16 i_extra_isize;
    __le16 i_pad1;
    __le32 i_ctime_extra; /* время модификации inode (наносекунды) */
    __le32 i_mtime_extra; /* время модификации данных файла (наносекунды) */
    __le32 i_atime_extra; /* время доступа (наносекунды) */
    __le32 i_crtime; /* время создания файла */
    __le32 i_crtime_extra; /* время создания файла (наносекунды) */
    __le32 i_version_hi; /* старшие 32 бита версии */
};

```

Размер дополнительной секции может меняться от версии к версии и хранится в поле `i_extra_isize`, следующем сразу за старой 128-битной частью. Суперблок содержит 2 связанных с этим вопросом поля: `s_min_extra_size`

(гарантированный размер дополнительной статической области) и `s_want_extra_size` (желаемый некоей версией, но не гарантируемый размер экстра-области).

Оставшееся пространство в `inode` может быть использовано для хранения встроенных расширенных атрибутов прямо в `inode`, что существенно увеличивает производительность их обработки. Также по-прежнему доступен дополнительный EA-блок, позволяющий хранить еще 4 Kb атрибут-данных для каждого файла. Возможность хранения большего количества EAs в форме регулярного каталога не реализована.

### **Предразмещение (preallocation)**

Некоторым приложениям, например базам данных и потоковым медиа-серверам, необходимо предразмещать "впрок" блоки для расширения файла, но без записи в них каких-либо данных. Предразмещение позволяет выделять блоки как можно более непрерывно и гарантирует необходимое пространство для записи в пределах предразмещенной области. Внутренне файловая система рассматривает неинициализированные участки файла как заполненные нулями, что позволяет избежать несанкционированного просмотра устаревших данных. Предразмещение должно быть устойчивым к перезагрузке, в отличие от так называемого резервирования блоков в `ext3/4`.

В `ext4` предразмещение реализовано вполне традиционно – неинициализированный экстенд имеет специальную пометку, обнаружив которую при попытке чтения экстенда, файловая система вернет приложению блок нулей. При записи в середину такого экстенда он разбивается на две части – одна дополняется нулями и пишется на диск, другая остается "виртуальной".

В настоящее время все файловые системы Linux, располагающие соответствующим функционалом, принимают запросы на предразмещение через `ioctl()`. В будущем (по скольку таких ФС становится все больше) планируется введение специального системного вызова, реализующего `posix_fallocate` API.

### **Отложенное и многоблочное размещение**

Аллокатор `ext3` может размещать только по одному блоку за раз, что не эффективно при высокой интенсивности ввода-вывода. Т.к. запросы на выделение блоков передаются на уровень VFS по одному за раз, аллокатор `ext3` не может предвидеть будущие запросы и кластеризовать их, что так же негативно сказывается на фрагментации ФС.

Отложенное размещение – хорошо известная техника, суть которой заключается в отсрочке выделения блоков до времени сброса страниц (`flush`). Это позволяет обеспечить более эффективную, с точки зрения фрагментации и нагрузки на CPU, группировку запросов на размещение. Короткоживущие временные файлы при этом могут вообще не получить дискового воплощения, оставаясь лишь в кэше. Патчи с реализацией отложенного размещения для `ext4`

уже написаны, сейчас идет работа по выносу соответствующего функционала на уровень VFS для его разделения с другими файловыми системами.

С появлением отложенного размещения стало возможно реализовать т.н. многоблочное размещение, при котором дисковое пространство выделяется сразу целыми экстендами, что исключает множество лишних вызовов `ext4_get_blocks()` и `ext4_new_blocks()` и уменьшает нагрузку на процессор.

Многоблочное размещение в `ext4` реализовано через сбор информации о свободных экстендах в каждой группе блоков при монтировании ФС и организации ее хэширования в оперативной памяти.

Прирост производительности от отложенного и многоблочного размещения оказался очень существенным – на 30% возросла пропускная способность ФС, нагрузка на CPU снизилась почти на 50%. Цена этого решения – увеличение времени монтирования ФС.

Кроме прочего, в разработке находятся еще две возможности, надстраиваемые над отложенным многоблочным размещением, с которыми связывают большие надежды по серьезному уменьшению фрагментации ФС:

- Перспективное предразмещение (`in-core preallocation`) – информацию о свободных экстендах можно использовать для построения более мощного механизма предразмещения и резервирования дискового пространства. Каждый `inode` может иметь несколько заранее зарезервированных сегментов, индексируемых в логических блоках, что, к примеру, облегчит НПС-приложения запись в файл со множества узлов по совершенно не предсказуемым смещениям.
- Группы размещения (`locality groups`) – в настоящее время решение по выделению блоков принимается независимо для каждого файла. Однако, если аллокатор будет располагать информацией о логических взаимосвязях объектов ФС, он сможет размещать связанные файлы ближе друг к другу, что существенно улучшит производительность чтения/поиска. Аллокатор может отслеживать некоторое количество еще невыделенных блоков (на уровне групп) и попытаться зарезервировать для них соответствующее количество дискового пространства. Этот объем будет использован позднее, при сбросе страниц на диск, для назначения конкретных блоков конкретным файлам. Все это, ценой незначительного повышения нагрузки на CPU, приведет к существенному уменьшению фрагментации файловой системы и более оптимальному, с точки зрения производительности, размещению связанных данных на диске.

Можно с уверенностью сказать, что `ext4` располагает достаточно мощным механизмом распределения дискового пространства, соответствующим современным требованиям к эффективной обработке больших и маленьких дисковых запросов под многопоточными нагрузками.

## **Фоновая дефрагментация**

Хотя функционал, описанный выше, существенно улучшает фрагментационную устойчивость файловой системы, со временем, на активно используемой ФС, фрагментация все же может достичь существенных значений. Для решений этой проблемы была разработана программа e4defrag, способная дефрагментировать как отдельные файлы, так и весь том. При работе на отдельным файлом программа создает временный inode и выделяет под все данные файла один (если возможно) или несколько экстенгов, используя многоблочное размещение. После копирования данных файла блочные указатели в оригинально inodes подменяются на новые, и временный inode удаляется.

## **Надежность**

Надежность хранения данных является одной из самых важных характеристик ext3, и, пожалуй, главной причиной ее популярности. Стараясь сохранить эту репутацию, разработчики ext4 приложили много усилий для обеспечения надежности файловой системы. Не смотря на использование журналирования и различных RAID-конфигураций, многие файловые системы подвержены различным разрушениям дисковой структуры. Поэтому первой линией защиты целостности данных является проактивное обнаружение проблем, комбинирующее устойчивый формат хранения, избыточность на разных уровнях и проверка целостности с помощью контрольных сумм.

Одна из важнейших характеристик любой файловой системы – время восстановления после сбоя. К примеру, на вполне обычном в наше время массиве объемом в 2 Tb ext3 будет восстанавливаться до работоспособного состояния, в среднем, от двух до четырех часов, а в худшем случае – до нескольких суток. Использование экстенгов очень хорошо сказалось на масштабировании этого параметра, в разработке также находятся еще несколько дополнительных схем.

## **Подсчет неиспользуемых inodes и ускорение e2fsck**

Безусловно, наиболее трудоемкой операцией в e2fsck является проверка inodes на первом проходе. Она требует чтения с диска всех таблиц inodes, поиск в них целых, разрушенных и неиспользуемых inodes, внесение необходимых исправлений и обновление битовых карт размещения блоков и inodes. Неиспользуемые группы блоков и таблицы inodes теперь имеют особую логическую метку, позволяющую пропустить значительную часть первого прохода, что позволит существенно уменьшить общее время работы e2fsck. Эта возможность может задействована во время mke2fs или позднее, с помощью опции -O uninit\_groups утилиты tune2fs.

С этой опцией ядро сохраняет количество неиспользуемых inodes в конце inode-таблицы каждой группы блоков. В результате e2fsck может пропустить как чтение этих блоков с диска, так их сканирование на предмет поиска разрушений. Для подтверждения подлинности информации о неиспользуемых inodes дескриптор группы содержит контрольную сумму (CRC16).

Т.к. типичная ФС типа ext3 использует от 1 до 10 процентов своих inodes и все они находятся в началах inode-таблиц, указанное нововведение позволяет избежать чтения и обработки существенного количества метаданных на первом проходе e2fsck. В настоящее время драйвер ext4 не увеличивает счетчик неиспользованных inodes при удалении файлов – это делает только e2fsck. Поэтому на ФС с большим количеством удаленных файлов fsck будет работать значительно быстрее при втором запуске.

## **Контрольные суммы**

Введение контроля целостности метаданных с помощью контрольных сумм позволяет ext4 быстрее и достовернее отыскивать разрушения и соответственно на них реагировать, вместо того, что бы слепо верить прочитанным с диска данным. Дескриптор группы блоков в ext4 также защищен контрольной суммой, в планах на ближайшее время стоит защита журнала, поскольку плотность хранения важных метаданных в нем очень высока, что существенно повышает риск разрушения ФС в результате повреждения журналируемых данных из-за аппаратной ошибки. Соответствующий патч в настоящее время почти готов к внесению в код ext4.

Каждая транзакция имеет блок-заголовок и фиксирующий блок (commit-block). Во время нормальной журнальной операции фиксирующий блок не сбрасывается на диск, пока там не окажется заголовок и все метаданные. Следующая транзакция будет вынуждена ждать полной фиксации предыдущей, прежде чем начинать изменение ФС. Таким образом, совпадение номера транзакции у заголовка и фиксирующего блока служит сигналом к запуску этой транзакции во время восстановления ФС, а если номера различаются – чтение журнала прекращается. Однако существуют несколько сценариев, в которых подобное поведение может вызвать разрушение метаданных.

В защищенном журнале драйвер ext4 вычисляет контрольную сумму (CRC32) для всех блоков транзакции (включая заголовок), и записывает ее в фиксирующий блок. Если, при восстановлении из журнала, обнаруживается несоответствие этой контрольной суммы фактически прочитанным с диска данным, это значит, что один или несколько блоков метаданных оказались разрушены или вовсе не были записаны на диск в результате ошибки оборудования. В таком случае транзакция отбрасывается целиком, а пользователю рекомендуется воспользоваться e2fsck.

Также значительным выигрышем от использования подобной защиты журнала является отказ от пошаговой фиксации транзакции (запись фиксирующего блока теряет всякий смысл), что может увеличить пропускную способность журнала более чем на 20%.

В долгосрочной перспективе обсуждается возможность добавления защиты контрольными суммами для экстендов, битовых карт, inodes и, возможно, даже каталогов. С появлением контрольных сумм в журнале все это выполняется введением лишь небольшого количества дополнительной логики – ведь все эти структуры при модификации обязательно проходят через журнал (нельзя не заметить, что от ошибок диска на местах постоянного пребывания этих данных такая схема никоим образом не защищает).

## **Другие нововведения**

Разработчики продолжают постоянно добавлять новые возможности в ext4. Два наиболее ожидаемых из всех нововведений заключаются в реализации наносекундных временных штампов и отслеживание версий inode (versioning). Эти две возможности позволяют достичь абсолютной точности при работе с временем доступа к файлу и отслеживании изменений в нем.

Ext3 располагает тайм-штампами второго порядка, однако на современных высокопроизводительных процессорах и системах хранения этого недостаточно. С введением увеличенного inode в ext4 появилась возможность записывать временные штампы с наносекундной точностью, на что достаточно 30 бит 32-битного поля штампа. Оставшиеся 2 бита применяются для расширения эпохи на 272 года.

Клиенты 4-й версии NFS нуждаются в возможности обнаруживать изменения в файле, сделанные на стороне сервера, что бы поддерживать клиентский кэш в актуальном состоянии. Для этого в ext4 на каждый inode введен глобальный 64-битный счетчик, инкрементируемый драйвером после каждого обновления файла. Сравнивая значения счетчика на стороне сервера со своим, клиент NFSv4 может понять, изменялся ли файл. Этот счетчик обнуляется при создании файла, а его переполнение не имеет значения, т.к. проверяется лишь равенство.

## **Миграция на ext4**

В свое время разработчики уделили особое внимание сохранению обратной совместимости ext3 с ext2, и пользователи очень оценили эту возможность. Однако, в случае с ext4, ряда несовместимых с ext3 изменений дискового формата избежать не удалось, хотя пользователи могут исключительно просто обновить свою ext3 до новой версии, также, как это было при переходе с ext2. Доступны методы как для простого ознакомления с возможностями новой ФС, так и для миграции всей файловой системы без применения резервных копий.

Существует очень простой способ обновления ext3 для использования экстенгов и других возможностей ext4 без каких либо изменений на диске. Достаточно просто смонтировать существующую ext3 как ext4, и вновь созданные файлы будут размещаться в экстенгах, в то время как старые продолжат обрабатываться через модуль косвенной адресации. Особый флаг в inode позволяет драйверу различать формат inode и поддерживать их сосуществование в одной файловой системе. Все новые возможности ext4, основанные на применении экстенгов (предразмещение, многоблочное размещение), будут немедленно доступны для вновь созданных файлов.

Кроме того, для полной миграции с ext3 на ext4 со временем будет подготовлена специальная утилита, осуществляющая преобразование косвенной структуры адресации в экстентную, и увеличивающая размер inode до 256 байт. В качестве частичной меры переход к экстенг-адресации может быть выполнен

достаточно просто – с использованием процедуры дефрагментации, во время которой файлы не только перемещаются в новые экстенды, но и дефрагментируются. Эта процедура может работать на смонтированной ФС.

Полный переход к ext4, включая преобразование inodes, осуществляется только на отмонтированной ФС, путем сканирования всей файловой системы, переформатирования inodes и перехода к экстенд-адресации.

Откат с ext4 обратно к ext3 также возможен, хотя и не столь очевиден и красив, как обновление ФС. Для этого необходимо смонтировать раздел с опцией -поextents, скопировать все файлы во временные, а затем заменить ими оригинальные. После того, как все файлы будут приведены к косвенной схеме адресации, необходимо, с помощью tune2fs, сбросить флаг INCOMPACT\_EXTENTS, а затем смонтировать ФС как ext3.

Для выполнения полной миграции на ext4 с файловых систем предыдущих поколений (ext2/3) разработчики, не смотря на признанную стабильность кода новой ФС, рекомендуют сделать резервную копию важных данных с обновляемого раздела. Т.к. обновленный раздел более не удастся примонтировать как ext2/3, необходимо убедиться, что вы располагаете необходимым набором ПО для полноценного использования новой файловой системы:

- e2fsprogs 1.41.6
- mount 2.16
- ядро Linux от 2.6.28 и выше
- grub 1.96+20090808 (если обновляется загрузочный раздел)

Если обновляется раздел ext2, то для начала нужно конвертировать его в ext3:

```
sudo tune2fs -j [device]
```

Перед началом обновления до ext4 рекомендуется сделать проверку файловой системы на наличие ошибок:

```
fsck.ext3 -pfy [device]
```

Затем включаем все предоставляемые ext4 новые возможности:

```
tune2fs -O extents,uninit_bg,dir_index [device]
```

После чего также рекомендуется пройтись по разделу утилитой fsck:

```
fsck.ext4 -yfpD /dev/sdc1
```

Опция -D заставит fsck заняться оптимизацией каталогов – для директорий с поддержкой индексов будет выполнено переиндексирование, для обычных линейных каталогов ext3 – пересортировка и сжатие. Опция -r отвечает за проверку ошибок, -u заставит утилиту автоматически отвечать согласием на все возникающие в ходе проверки вопросы, -f – принудительная проверка даже в случае, когда файловая система выглядит чистой.

### ext4 на фоне предшественников и конкурентов

Посмотрим на сравнительную таблицу основных характеристик ext3, ext4 и XFS (как наиболее производительной и функциональной, по мнению автора, из современных ФС):

	ext3	ext4	XFS
Предельный размер раздела	16 Tb	1 Eb	16 Eb
Предельный размер файла	2 Tb	16 Tb	8 Eb
Количество файлов	2 <sup>32</sup>	2 <sup>32</sup>	2 <sup>64</sup>
Размер inode	128 байт	256 байт	переменный
Учет занятых блоков	косвенно-блочная схема	экстенты	экстенты
Учет свободных блоков	битовые карты	битовые карты	B+ деревья
Разрешение временных штампов	секунды	наносекунды	секунды
Количество подкаталогов	2 <sup>16</sup>	не ограничено	не ограничено
Объем под Extended Attributes	4 Kb	>4 Kb	не ограничен
Дефрагментация	отсутствует	реализована	реализована
Индексирование каталогов	отсутствует	реализовано	B+ деревья
Отложенное размещение	отсутствует	реализовано	реализовано
Многоблочное размещение	элементарное	продвинутое	продвинутое

### Мнение автора

Как видно из приведенной выше таблицы, ext4 уступает по многим параметрам даже разработанной более 15 лет назад XFS, не говоря уже о файловых системах нового поколения – ZFS и btrfs. И все же она остается идеальным выходом для системных администраторов, годами использующих ext3 как наиболее надежную и хорошо поддерживаемую файловую систему с богатым набором средств устранения неполадок и восстановления поврежденных данных. Довольно легко обновив существующие ext2/3 разделы до новой версии, пользователи получают для своих хранилищ достаточно производительную и очень надежную ФС, не имеющую каких-либо явных ограничений, типа максимального размера раздела или файла.

## **Источники**

1. Исходные тексты ядра Linux версии 2.6.32
2. <http://ext4.wiki.kernel.org>
3. Ext4 block and inode allocator improvements  
(<http://ols.fedoraproject.org/OLS/Reprints-2008/kumar-reprint.pdf>)
4. The new ext4 filesystem: current status and future plans(<https://ols2006.108.redhat.com/2007/Reprints/mathur-Reprint.pdf>)