author: Пешеходов А. П. aka fresco (fresco_pap@mail.ru)
mtime: 31.01.2007

Файловая система FreeBSD

Статья основана на докладе Marshall Kirk McKusick на конференции BSDCon 2003, оригинальный текст которого доступен по адресу:
www.usenix.org/events/bsdcon03/tech/full_papers/mckusick/mckusick.pdf

Операционная система FreeBSD всегда отличалась немного консервативным подходом к организации хранения данных. В то время, как Linux предлагает пользователям широкий спектр файловых систем, основанных на новейших алгоритмах и исследованиях, разработчики FreeBSD упорно продолжает совершенствовать свою UFS, разменявшую недавно третий десяток. Попробуем разобраться, что ими движет.

Немного истории

Первая версия FFS – быстрой файловой системы Berkley, более известная пользователям FreeBSD как UFS1, появилась в 1984 году с выходом ОС 4.2BSD, откуда и была унаследована или позаимствована FreeBSD, OpenBSD, NetBSD, Sun Solaris и многими другими вариациями UNIX. Со временем те или иные производители вносили изменения в код ФС для устранения "узких мест" и придания новых возможностей: так фирма Sun Microsystems реализовала журналирование для своей версии FFS, команда NetBSD самостоятельно выполнила аналогичную задачу, выпустив так называемую Log Structureg Filesystem – LFS (этот код достался, после разделения проектов, также и OpenBSD), разработчики FreeBSD внесли множество усовершенствований в свою UFS2. Не стоит забывать, что и ext2, "родная" файловая система Linux, разрабатывалась на основе FFS. Не смотря на то, что архитектура UFS1 достаточно подробно описана во множестве источников, считаю необходимым коротко на ней остановиться.

Кратко о UFS1

Файловая система UFS1 состоит из следующих частей:

- 8-Кб загрузочного блока
- суперблока, содержащего magic-идентификатор ФС, а также параметры геометрии, статистики и текущей настройки ФС.
- набора групп цилиндров (cylinder groups, CG)

Изначально группа цилиндров содержала в себе один или более последовательных

цилиндров физического диска, однако сейчас эта структура используется только в качестве удобного способа управления логически близкими группами блоков.

Каждая CG содержит резервную копию суперблока, пространство для inodes (по умолчанию размещается 1 inode на 4 фрагмента ФС), битовую карту inodes, битовую карту блоков и статистические сведения – размер CG и количество свободных блоков и inodes в ней. Битовая карта блоков, принадлежащих группе цилиндров, должна помещаться в 1 блок файловой системы. Исходя из этого утилита newfs вычисляет минимальное количество групп цилиндров на раздел при его форматировании. Для типичного размера блока UFS1 в 16 Кб это 6 групп цилиндров на 1 Гб дискового пространства.

Дисковый inode – основной контейнер метаданных файла, имеет следующую структуру (src/sys/ufs/ufs/dinode.h):

```
struct ufs1_dinode {
      u_int16_t   di_mode;          /* Разрешения и тип файла */
      int16_t     di_nlink;         /* Количество ссылок */

      union {
            u_int16_t oldids[2];    /* Старые ID пользователя и группы (FFS) */
            } di_u;

      u_int64_t   di_size;          /* Размер файла в байтах */
      int32_t     di_atime;         /* Время последнего доступа */
      int32_t     di_atimensec;     /* Наносекунды */
      int32_t     di_mtime;         /* Время последнего изменения файла */
      int32_t     di_mtimensec;     /* Наносекунды */
      int32_t     di_ctime;         /* Время последнего изменения inode */
      int32_t     di_ctimensec;     /* Наносекунды */
      ufs1_daddr_t      di_db[NDADDR];    /* 12 указателей на прямые блоки */
      ufs1_daddr_t      di_ib[NIADDR];    /* 3 указателя на косвенные блоки */
      u_int32_t   di_flags;         /* Флаги */
      int32_t     di_blocks;        /* Количество выделенных блоков */
      int32_t     di_gen;           /* Поколение */
      u_int32_t   di_uid;           /* ID Владельца */
      u_int32_t   di_gid;           /* ID группы */
      int32_t     di_spare[2];      /* Зарезервировано */
};
```

Каталог UFS1 состоит из из некоторого количества элементов каталога (src/sys/ufs/ufs/dir.h):

```
struct direct {
      u_int32_t   d_ino;                /* Номер inode */
      u_int16_t   d_reclen;             /* Длина элемента каталога */
      u_int8_t    d_type;               /* Тип файла */
      u_int8_t    d_namlen;             /* Длина имени */
      char        d_name[MAXNAMLEN + 1];  /* Строка с именем файла <= MAXNAMLEN */
};
```

Как обычно, элемент каталога выполняет роль связующего звена между именем файла и номером его inode.

Предпосылки к созданию UFS2

Традиционная BSD FFS и многие ее производные использовали 32-битные указатели на дисковые блоки, принадлежащие файлам. UFS1 была спроектирована в середине 80-х годов прошлого века, когда самые большие диски были размером около 500 Мб. В то время обсуждалось, стоит ли тратить 32 бита на указатель, или оставить его 24-битным. К счастью, футуристический взгляд одержал верх и в архитектуре FFS использовались 32-битные указатели. За 20 лет применения UFS1 системы хранения данных выросли в объемах до многих терабайт. В зависимости от размера блока 32-битный указатель UFS1 исчерпывает свои возможности на разделе размером от 1 до 4 Тб. Программисты команды FreeBSD долгое время полагали, что проблему с размером раздела можно решить некоторыми временными мерами, однако в 2002-м окончательно стало ясно, что единственное долгосрочное решение – это 64-битные указатели. Таким образом, было решено создать новую файловую систему – UFS2.

Разработчики рассматривали и альтернативные пути: от внесения дальнейших улучшений в FFS до импортирования сторонней файловой системы (например XFS или reiserfs). Также обсуждался вариант написания новой файловой системы с нуля – с учетом последних исследований в этой области и опыта применения FFS. Вариант доработки UFS1 был выбран потому, что он позволил на базе существующего кода быстро разработать и внедрить устойчивую и надежную файловую систему, а также использовать одну и ту же основу для поддержки и UFS1, и UFS2. 90 процентов кода разделяются драйверами этих ФС, поэтому практически любое изменение (будь то bugfix, новая возможность или улучшение производительности), как правило, относится к обеим файловым системам.

Файловая система UFS2

Дисковый inode, используемый UFS1, имеет размер 128 байт и только 2 неиспользуемых 32-битных поля. Было бы невозможно перейти к 64-битным указателям без сокращения количества прямых указателей на блоки с 12 до 5. Однако этот шаг серьезно увеличил бы потери дискового пространства. Таким образом, единственным выходом стало увеличение размера дискового inode до 256 байт.

Кроме того, при переходе к новому формату дисковых inodes появилась возможность внести множество связанных усовершенствований, которые невозможно

было использовать в UFS1. Конечно, соблазнительно было внести все улучшения, предлагавшиеся за прошедшие 20 лет, однако программисты решили, что следует ограничить поток новых возможностей теми из них, что будут действительно полезны. Каждое дополнение добавляет сложность, которая может сказаться как на скорости его внесения в код, так и на производительности файловой системы. Нечеткие или редко используемые возможности могут увеличить количество проверок в часто вызываемых функциях и, таким образом, ухудшить общую производительность ФС даже если и вовсе не используются.

Хотя разработчики и приняли решение перейти к новому формату inode, изменение структур суперблока, групп цилиндров и директорий было нежелательно. Дополнительная информация, необходимая для суперблока и групп цилиндров UFS2, хранится в ранее зарезервированных полях соответствующих структур UFS1. Сохранение одинакового формата этих структур для обеих файловых систем позволяет использовать одну кодовую базу для UFS1 и UFS2. Это избавляет ФС от необходимости проверять, с каким именно типом суперблока, группы цилиндров или элемента каталога она работает. Для минимизации количества проверок дисковые inodes обоих форматов преобразуются при чтении к единому виду, а при записи – обратно. Эффектом этого решения является наличие лишь девяти из нескольких сотен процедур, специфичных для UFS1 или UFS2. Выгода от использования единого кода для обеих файловых систем – значительное уменьшение стоимости сопровождения. Вне девяти чувствительных к формату файловой системы функций разработчики одним движением устраняют ошибки сразу в двух драйверах. Общий код также означает, что поддержка каких-либо новых возможностей ОС (таких, как SMP) добавляется только один раз для семейства файловых систем UFS.

Не смотря на то, что в обеих файловых системах все еще используются одинаковые структуры данных для описания групп цилиндров, в UFS2 их определение изменилось. Во времена UFS1 файловая система могла получить точное представление о геометрии диска, включая границы дорожек и цилиндров, и точно вычислить круговое положение каждого сектора. Современные диски скрывают эту информацию, предоставляя фиктивные количества блоков на трек, треков на цилиндр и цилиндров на диск, позволяющие, разьве что, вычислять размеры разделов. В современных машинах “диск”, предоставляемый файловой системе, в действительности может быть составлен из нескольких физических дисков RAID-массива. Хотя и проводились некоторые исследования по выяснению истинной геометрии диска, сложность эффективного использования такой информации очень высока. Треки на внешней и внутренней кромках современных дисков имеют разное число секторов, и это делает вычисление круговых координат любого конкретного сектора очень сложным. Поэтому в UFS2 было решено избавиться от кода, привязанного к геометрии диска и круговым координатам, и просто хранить файлы в логически близких друг к другу блоках, что дало даже большую производительность. Старый код обработки групп цилиндров начал удаляться из USF1 в

конце 80-х, однако полностью избавиться от него удалось только при разработке UFS2.

Файловая система UFS1 использует 32-битные номера inodes. Хотя было соблазнительно увеличить разрядность номера inode до 64 бит, это привело бы к изменению формата директории. Существует очень много кода, работающего непосредственно (а не через интерфейсы) с элементами каталогов.

Изменение формата каталога привело бы к созданию гораздо большего количества специфичных для UFS2 функций, которые увеличили бы сложность сопровождения кода. Кроме того, текущие API используют 32-битные номера inodes для работы с элементами каталогов. Так, даже если бы файловая система поддерживала 64-битные inodes, в настоящее время они были не доступны для приложений. Современные системы даже близко не подходят к ограничению в 4 миллиарда файлов на раздел, налагаемому 32-битным номером inode. Экстраполируя статистику роста количества файлов на разделах за прошедшие годы, можно смело предположить, что 32-битного номера inode будет достаточно еще 10-20 лет. На будущее в суперблоке UFS2 зарезервировано место под флаг, указывающий, что файловая система имеет 64-битные номера inodes.

Также обсуждалась возможность введения более сложной структуры каталога с использованием бинарных структур (B+деревьев) для ускорения операций с большими директориями. Эта технология используется многими современными файловыми системами – XFS, JFS, reiserfs и ext4. Разработчики решили не вносить это усовершенствование по нескольким причинам. Во-первых, они были ограничены во времени и ресурсах, в то время, как стояла задача получить стабильно работающую ФС к моменту выпуска FreeBSD 5.0-RELEASE. Сохранив старый формат каталога, программисты смогли использовать весь код работы с директориями из UFS1 и избежали необходимости переписывать различные ФС-утилиты. Еще одна причина, по которой было решено сохранить старый формат директорий – динамическое хэширование каталога, которое усовершенствует схему индексирования директории. Что бы избежать повторения линейных поисков в больших каталогах, механизм динамического хэширования при первом обращении к директории на лету строит хэш-таблицу элементов каталога. Эта таблица позволяет избежать сканирования каталога при последующих операциях поиска, создания и удаления. В отличие от файловых систем, изначально спроектированных для работы с большими каталогами, в UFS2 эти структуры не сохраняются на диске, и поэтому ФС обратно-совместима. Благодаря технике динамического хэширования разработчикам удалось существенно снизить влияние больших каталогов на производительность UFS2.

По образцу ext3 был введен флаг, показывающий, поддерживаются или нет дисковые структуры индексирования для каталогов. Этот флаг безоговорочно сброшен в текущей реализации UFS2. В будущем, когда поддержка дисковых структур индексации каталогов будет реализована, драйвер ФС не будет сбрасывать этот флаг. Новая ФС

прочитает и использует индексы, оставив флаг установленным. Старая же система, смонтировав однажды этот раздел, сбросит флаг; поэтому, когда управление разделом вновь будет передано новому драйверу, он обнаружит, что индексы не использовались по крайней мере некоторое время и должны быть пересозданы заново. Единственным ограничением в реализации индексов будет то, что они останутся лишь дополнительной структурой, ссылающейся на данные старого линейного формата.

Расширенные атрибуты (extended attributes)

Основное дополнение в UFS2 (относительно UFS1) – это поддержка расширенных атрибутов. Extended attributes (EA) – это часть дополнительного хранилища данных, связанного с inode и используемого для хранения вспомогательных данных, отделенных от содержимого файла. Идея EA концептуально подобна потокам данных, используемым в HFS и HFS+ – файловых системах Apple MacOS. Интегрирование EA в inode позволяет гарантировать их целостность на уровне данных файла. Т.е. успешное завершение вызова fsync() означает, что данные файла, расширенные атрибуты и все имена и пути к файлу корректны и находятся на диске.

В inode UFS1 есть место для хранения двух блоков EA. Новый формат inode в UFS2 резервирует пространство для 5-ти дополнительных 64-битных указателей. Таким образом, количество блоков EA может быть от 1 до 5. В настоящее время выделено 2 блока для расширенных атрибутов, а 3 других оставлены как запас на будущее. Несмотря на это весь код подготовлен к обработке массива указателей, поэтому, когда количество расширенных атрибутов в будущем переполнит доступное пространство, существующий код будет работать без изменений. Сохраняя 3 блока в запасе, разработчики обеспечили достаточно место для будущего использования. Если пространства под EA понадобится совсем уж много, один из запасных блоков может быть использован для хранения косвенных ссылок на блоки, содержащие указатели на EA.

Формат расширенных атрибутов

Поля заголовка EA имеют следущие параметры:

Поле	Длина (байт)
length	4
name_space_class	1
content_pad_len	1
name_lenth	1
name	name_len

Содержимое атрибута выравнивается по 8-байтной границе, его размер дополняется

до величины, хранящейся в поле content_pad_len. Приложения, "не понимающие" некоторые типы атрибутов, могут пропускать их, добавляя длину к текущей позиции, что бы получить следующий атрибут. Таким образом, множество разных программ могут совместно использовать пространство EA, даже если "не понимают" используемых "коллегами" типов данных.

Первым из двух изначальных применений для EA была поддержка списков контроля доступа (Access Control List – ACL). ACLs заменяют традиционную для UNIX групповую схему доступа к файлу на более специфичную – с двумя списками: пользователей, имеющих доступ к файлу, и конкретных прав для каждого пользователя. Эти права включают как традиционные чтение, запись и исполнение, так и расширенные – право на переименование, удаление файла, и пр.

В ранних версиях ACLs были реализованы в виде единственного на всю ФС дополнительного файла, индексированного по номерам inodes и разбитого на части небольшого размера для хранения ACL-разрешений. Размер каждого такого отрезка был очень мал, т.к. ACL-файл резервировал место для каждого номера возможного inode. Эта реализация имела 2 недостатка. Фиксированный размер пространства, выделяемого каждому inode, означал, что длинна списка пользователей будет ограничена. Также было сложно атомарно обновлять изменения, вносимые в ACL файла, т.к. inode файла и блок ACL никак не могли быть записаны одновременно.

Обе проблемы этой реализации были решены хранением ACL-данных в EA-области inode. из-за большого размера EA-области (минимум 8 kb, типично 32 kb) стало легче хранить длинные ACL-списки. Атомарное обновление так же упростилось, т.к. запись на диск inode приводила к фиксации всех адресуемых им наборов данных (в том числе и EA-областей) в одной дисковой операции, в то время как старый ACL-файл обновлялся только при вызове fsync().

Новые возможности файловой системы

С введением нового формата inode появилась возможность внести несколько дополнительных усовершенствований. Разработчики решили заранее позаботиться о "проблеме 2038" (именно в 2038-м году переполнятся 32-битные поля, хранящие количество секунд, прошедших с начала 1970 года). Поля времен модификации, доступа и изменения были расширены до 64 бит. Рассматривалось расширение временных штампов только до 48 бит, однако разрядность в 64 бита показалось программистам оптимальной для большинства будущих CPU.

В тоже время было добавлено новое поля для хранения времени создания файла. Это поле устанавливается при первом размещении inode и не меняется вплоть до его

удаления. Оно было добавлено в структуру, возвращаемую вызовом stat() для тех приложений, которым может оказаться полезным это значение (к примеру, различные архивирующие программы, такие, как dump, tar и pax). Время создания было добавлено в ранее зарезервированное поле структуры stat, так что размер структуры не изменился. Таким образом, старые программы смогут также корректно обрабатывать результат вызова stat().

В настояще время только программа dump изменена для использования времени создания файла. Эта новая версия dump, которая может обрабатывать разделы как UFS1, так и UFS2, создает дамп нового формата, не читаемого старой версией restore.

Вызов utimes(), что устанавливает произвольные временные штампы на файл, используется, как правило, при разархивировании файлов для выставления корректных значений времен доступа. Благодаря использованию этих программ может оказаться, что время создания файла будет больше, чем время его модификации. Семантика этого вызова была изменена таким образом, что, хотя он и не позволяет вручную устанавливать время создания файла, но следит за тем, что бы оно никогда не принимало значения меньше, чем меньшее из времен модификации.

Еще одно изменение коснулось поля флагов inode, которое теперь разбито на 2 отдельных 32-битных поля. В первом поле содержатся доступные пользователю флаги, во втором – флаги ядра, такие, как SNAPSHOT или OPAQUE. Таким образом, флаги ядра больше не могут быть изменены вредоносным приложением.

Динамические inodes

Один из основных недостатков UFS1 состоит в том, что она размещает все свои inodes во время создания ФС. Для файловых систем с миллионами файлов процесс инициализации может занимать несколько часов. Кроме того, программа создания новой ФС, newfs, предполагает, что каждая файловая система будет заполнена большим количеством мелких файлов и размещает гораздо больше inodes, чем обычно требуется. Если UFS1 использовала все свои inodes, единственный способ получить больше – создать дамп ФС, переформатировать е и восстановить данные из дампа. UFS2 решает эту проблему динамическим размещением inodes. Обычно реализация динамического выделения inodes требует отдельных структур метаданных, отслеживающих текущие наборы inodes. Управление и поддержка этих структур добавляет сложности и отрицательно сказывается на производительности ФС.

Для ослабления влияния этих факторов UFS2 размещает некоторое количество номеров inodes в блоках каждой группы цилиндров. Изначально каждая CG имеет один блок заранее выделенных inodes (32 или 64 inodes). Когда это блок переполняется,

размещается и инициализируется следующий блок inodes. Блоки, которые могут быть выделены под inodes, не являются частью пространства прочих свободных блоков и отдаются под данные файлов только в том случае, если все другие блоки в ФС исчерпаны.

Теоретически ФС может заполнится, использовав все свободные inode-блоки. Позднее, когда будут удалены некоторые большие файлы и на их месте понадобится создать множество мелких, ФС может обнаружить, что не в состоянии разместить требуемое количество inodes, т.к. все inode-блоки заняты. Может понадобится переразместить существующие файлы для перемещения их данных в обычные свободные блоки и освобождения места под inodes. Этот код еще не написан, т.к. подобная ситуация вряд ли возникнет на практике, потому что резерв свободного места на подавляющем большинстве файловых систем (8%, напомню, что UFS жестко устанавливает этот предел, и если ФС заполнена данными ровно на 92%, вызовы, требующие размещения блоков, будут работать только от root, возвращая процессам других пользователей ENOSPC) больше требуемого для inodes пространства (2-6%). На таких системах только процессы, запущенные от root, имеют право размещать inode-блоки. Как только в этом коде появится практическая необходимость, он сразу же будет написан. Пока же ФС, при возникновении описанных условий, сообщает об исчерпании inodes при попытке создать файл.

Одно из выгодных отличий динамического размещения inodes состоит в том, что время создания UFS2 составляет около 1% от показателей UFS1. Недостаток – необходимость дисковой записи большого куска данных при создании каждого 64-го inode.

Загрузочные блоки

UFS1 резервирует 8 Кб в начале раздела под загрузочные блоки. Хотя это пространство и выглядит огромным в сравнении с 1 Кб, который выделяли ФС предыдущего поколения, с другой стороны современные загрузчики требуют все больше и больше места. Поэтому разработчики решили пересмотреть размер загрузочного кода в UFS2. Теперь boot-сектор раздела может иметь размер 0, 8, 64 (по умолчанию) и 256 Кб.

Изменения и усовершенствования в soft updates

Традиционно целостность файловой системы обеспечивается или применением синхронной записи для упорядочивания зависимых обновления данных, либо использованием упреждающего журналирования для атомарной группировки этих обновлений. Soft updates (позволю себе перевести этот термин как "мягкое журналирование"), альтернативный подход, есть реализация механизма, который отслеживает зависимости изменений метаданных и организует их фиксацию на диске таким образом, что ФС всегда остается в непротиворечивом состоянии. Использование мягкого журналирования избавляет от необходимости содержать выделенный дисковый

журнал или прибегать к синхронной записи – оба эти подхода отрицательно влияют на производительность.

Добавление к inode расширенных атрибутов заставило внести изменение в код soft updates с тем, что бы обеспечить целостность и этих структур данных. Как и с регулярными блоками (т.е. блоками, содержащими данные регулярных файлов), необходимо быть уверенным, что блоки данных и битовые карты, имеющие отношение к EA, зафиксированы до того, как связанные изменения попадут в дисковый inode. Soft updates так же обеспечивают фиксацию блоков данных EA при вызове fsync() на файле.

В связи с этим в текущей реализации soft updates были сделаны два важных усовершенствования. Задуманные изначально для UFS2, они автоматически попали и в UFS1, т.к. код этой части обе ФС разделяют.

Для пользователя удаление файла происходит очень быстро, однако в действительности процесс освобождения inode файла и возвращения всех его блоков в свободный список может занимать несколько минут. Пространство, занимаемое файлом в UFS2, не попадает в ФС-статистику до тех пор, пока все его блоки не окажутся физически освобожденными. Таким образом, пользователь может удалить несколько файлов, однако освобождение места на диске увидит далеко не сразу. Это может стать серьезной проблемой для программ типа кэша браузера: обнаружив нехватку дискового пространства, он начинает небольшими кусками удалять наиболее старые данне, периодически проверяя сколько освободилось места. И пока данные о действительно свободном пространстве дойдут до программы df, из кэша могут быть удалены и самые последние данные. Для решения подобных проблем механизм soft updates теперь поддерживает специальный счетчик, содержащий количество блоков, удерживаемых файлом в процессе удаления. При вызове statfs() это счетчик добавляется к количеству свободных блоков ФС. В результате свободное место на диске появляется сразу после вызова unlink() или завершения работы rm.

Второе изменение касается ложных сообщений о нехватке дискового пространства. Почти заполненная файловая система может отвечать на конкретные запросы о выделении блоков сообщениями о нехватке места, даже если команда df говорит об обратном. Это происходит потому, что soft updates не управляют дисковым пространством, которое удерживают файлы, находящиеся на стадии удаления.

Изначально эту проблему пытались решить, просто позволяя процессам ждать появления свободного места. Однако ждать зачастую приходится около минуты. Кроме того, что такое ожидание делает приложение невыносимо медленным, еще удерживается блокировка на vnode, что сказывается также на работе других процессов. Хотя эти условия и не приводят к тупиковой ситуации и снимаются в течение 1-2 минут, пользователи часто

думают, что система зависла и жмут на reset.

Для излечения этой болезни в UFS2 было решено кооперировать процессы, ожидающие освобождения блоков и заставлять их работать в помощь soft updates. Планировщик задач отдает кванты времени ожидающих процессов подсистеме soft updates, которая использует их для ускорения фиксации изменений. Т.о., чем больше процессов в системе ожидают выделения блока, тем быстрее soft updates завершит свою работу, Обычно дело решается в 1-2 секунды.

Проверка целостности большой файловой системы

Обычно после аварийного выключения системы программа fsck проходит по всем inodes и битовым картам, проверяя, какие из inodes и блоков реально используются, и в случае необходимости вносит исправления. Текущая реализация soft udates гарантирует непротиворечивость всех метаданных файловой системы, однако остаются 2 возможные несогласованности: отсутствие в битовой карте пометки об освобождении реально нигде неиспользуемого блока и завышенное количество ссылок у inode. Конечно, использовать ФС после сбоя полностью безопасно даже без запуска fsck, однако может потеряться некоторое количество дискового пространства. Поэтому была разработана новая версия fsck, способная работать в фоновом режиме на активной файловой системе для поиска и восстановления потерянных блоков и inodes.

В сочетании со снапшотами эта задача по "сбору мусора" не выглядит такой уж сложной. При работе в фоновом режиме fsck делает снимок файловой системы и в дальнейшем проверяет его как обычную ФС. Отыскав все потерянные ресурсы, fsck, через специальный системный вызов, извещает ФС о необходимых изменениях и удаляет снапшот.

Кроме того, фоновая проверка большой файловой системы требует много памяти, объем которой растет пропорционально размеру ФС. На каждый inode регулярного файла требуется 4 байта, на inode каталога – 40-50 байт, на блок данных – 1 бит. На типичном разделе UFS2 с 16-Кб блоком и 2-Кб фрагментом fsck нуждается как минимум в 64 Мб памяти на 1 Тб файловой системы. Память, необходимая для проверки inodes,благодаря динамическому размещению, зависит от их количества. В худшем случае, как замечают разработчики, fsck при проверке большого раздела может потребовать гораздо больше памяти, чем реально установлено на машине, а теоретически, может превзойти предельный объем, доступный при 32-битной адресации. Надежда здесь только на то, что с ростом объемов дисков они будут заполняться фалами музыки и видео все больших размеров, что несколько ограничит рост потребления памяти во время работы fsck.

Программисты FreeBSD понимают, что журналируемые файловые системы

обеспечивают гораздо более быстрое и экономичное восстановление ФС. По этой причине начата работа по добавлению класса GJOURNAL к GEOM и постепенному отказу от использования soft updates. Однако даже журналируемые ФС нуждаются в fsck на случай сбоев оборудования или ПО.

Производительность

Производительность файловой системы UFS2 приблизительно аналогична UFS1. Это обусловлено прежде всего значительной общностью кодовой базы обеих ФС, и тем, что они используют одинаковые алгоритмы. Однако не стоит забывать, что целью создания UFS2 было не кардинальное увеличение производительности, которая уже составляет 80-95% от пропускной способности дисковой подсистемы, а обеспечение поддержки многотерабайтных файловых систем и новые возможностей (таких как EA) без потери производительности.

Планы на будущее

Политика выделения блоков UFS стремиться размещать блоки файлов на диске как можно более непрерывно. Метаданные, описывающие большой файл, в результате состоят из косвенных блоков с множеством последовательных номеров блоков, которые попусту забивают память, т.к. для открытого файла UFS старается хранить все метаданные в RAM. В UFS2 объем требуемой памяти удвоился, т.к. указатели на блоки стали 64-битными. Для экономии системной памяти и дискового пространства современные файловые системы используют экстенты – структуры данных, состоящие из пары чисел (адреса начала экстента и его длины), полностью описывающих непрерывный участок выделенного дискового пространства. В случае, когда файл размещен почти непрерывно, достигается существенная экономия системных ресурсов, однако если файл сильно фрагментирован, такой подход создаст еще больше проблем, чем традиционные косвенные блоки. Также стоит обратить внимание, что стоимость произвольного доступа к данным файла при использовании экстента существенно возрастает – может возникнуть необходимость перебрать множество экстентов для достижения нужного логического смещения в файле. В современных файловых системах (таких, как XFS, JFS, reiser4, ext4) эта проблема решается только с помощью индексирования экстентов по логическому смещению с помощью B+деревьев.

Разработчики FreeBSD только планируют внедрение схемы учета блоков файла на экстентах. А пока в inode добавлено поле, которое позволяет ФС использовать для конкретного файла больший размер блока. Для маленьких, слабо растущих или сильно фрагментированных файлов в этом поле устанавливается единица. А когда ФС обнаруживает большой непрерывный файл, она пишет сюда число от 2 до 16, на которое домножается размер блока ФС при работе с метаданными этого файла. Недостаток

подхода очевиден – исчерпав блоки большого размера, файловая система будет возвращать ENOSPC при попытке увеличить такой файл. Возможно, будет написан код, пересчитывающий все метаданные такого файла для уменьшенного размера блока. Эта процедура будет вызывать длинную паузу, однако предполагается, что подобные экстремальные условия будут возникать крайне редко.

Источники

1. "Enhancements to the Fast Filesystem To Support Multi-Terabyte Storage Systems", Marshall Kirk McKusick, BSDCon 2003, www.usenix.org/events/bsdcon03/tech/full_papers/mckusick/mckusick.pdf
2. en.wikipedia.org/wiki/UFS2
3. Исходные тексты ядра FreeBSD-6.1
4. "FreeBSD: Архитектура и реализация", Маршалл Кирк МакКузик, Джордж В. Невилл-Нил, изд. КУДИЦ-ОБРАЗ, Москва, 2006 г.

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