translator: Пешеходов Андрей aka fresco (filesystems@nm.ru)
released: 10.02.2008
modified: 10.02.2008

Файловая система BeOS

Введение

Данная статья является авторским переводом некоторых частей книги Dominic Giampaolo о файловой системе BeOS [1], причем главы для перевода выбирались по принципу наибольшей информативности в отношении структуры и алгоритмов BFS. Текст переведен не дословно, в том числе, участки исходного кода были заменены и дополнены аналогичными из исходников ОС Haiku — т.к. сама BeOS остается закрытым продуктом, а драйвер BFS для Linux (там она именуется befs) значительно устарел.

Если читателю покажется, что какие-то моменты освещены не достаточно подробно или вовсе пропущены, он может без проблем ознакомится с ними в оригинальной книге — благо написана она очень доступно и, можно сказать, увлекательно — словом, работать с ней было интересно!

Все упомянутые в статье файлы расположены в каталоге src/add-ons/kernel/file_systems/bfs дерева исходных текстов Haiku.

История

Be Inc. анонсировала BeOS с компьютером BeBox в октябре 1995-го, и в течение следующего года эти продукты были доступны для ознакомления разработчикам. Почти сразу стали видны многие проблемы, к примеру — наличие в системе множества различных баз данных, используемых для хранения разнородной пользовательской и системной информации, и существующих по верх достаточно простой ФС (Old File System, т.н. OFS). Разработчики намеренно допустили такую ситуацию, т.к. хотели вынести как можно больше кода в пользовательское пространство. Но, как оказалось, в следствие этого сильно пострадали надежность и производительность подсистемы хранения данных. Кроме того, в подобную концепцию (уровень VFS предельно простой архитектуры в ядре, основная функциональность — в user-space) не вписывалось желание поддерживать сторонние файловые системы (ISO9660, FAT и др.), для эффективной работы которых был необходим достаточно развитый API уровня

ядра.

Весной 1996 Be Inc. приступила к портированию BeOS на машины архитектуры PowerPC производства Apple, и появилась еще одна проблема — необходимость поддержки HFS. В результате, в сентябре 1996 года, Cyril Meurillon и Dominic Giampaolo начали работу над проектированием новой архитектуры ввода/вывода и уровня VFS для BeOS. Попутно было решено создать новую, высокопроизводительную файловую систему с интегрированной функциональностью СУБД.

Задача разделилась на 2 достаточно независимых компонента: создание высокоуровневого интерфейса для ФС и устройств (определение API для ФС и драйверов устройств, управление пространством имен, блокированием, состояниями объектов) и создание самой файловой системы с функциональностью, необходимой остальной части BeOS. Cyril, будучи ведущим архитектором ядра в Be Inc., взялся за выполнение первой задачи, а Dominic приступил к разработке ФС — т.е. определению ее дисковой структуры и алгоритмов обработки I/O запросов.

Разработка BFS полностью завершилась в начале 1997 года, и летом она была представлена общественности.

Структуры данных BFS

Что такое диск

BFS работает с блоками размером 1, 2, 4 и 8 кб. Блок в 512 байт не позволителен потому, что некоторые важные структуры ФС могут иметь больший размер, и их обработка усложнила бы реализацию блочного кэша и журналирования. Максимальный размер блока BFS будет обоснован ниже, т.к. необходимо понимание устройства некоторых структур данных до этого.

Важно понимать, что размер блока ФС не зависит от размера диска (это так практически во всех файловых системах, исключая Apple HFS, и, в некоторых случаях, FAT32). Выбор размера блока должен основываться на типе данных, которые будут храниться на создаваемом разделе: при большом количестве мелких файлов и размере блока в 8 кб будет теряться впустую слишком много дискового пространства, а ФС, хранящая большие файлы и имеющая блок размером 1 кб будет сильно подвержена фрагментации.

Как управлять дисковыми блоками

Существует несколько разных подходов к управлению свободным пространством на диске, самый простой — битовые карты. Другие методы — экстенты и B+деревья — достаточно сложны, поэтому в BFS применена bitmap-схема.

Основной ее недостаток — поиск больших протяженных участков свободного дискового пространства требует длительного линейного перебора множества битовых карт (избыточный расход места в свете объемов современных жестких дисков пренебрежимо мал). Причиной же выбора разработчиками BFS именно этой схемы послужили простота реализации и линейность зависимости (время)/(заполнение карты) для поиска.

Битовые карты хранятся на диске в виде простого массива байт, начинающегося в первом блоке. Место под них выделяется сразу при создании файловой системы.

Allocation groups

Группы размещения (allocation groups, далее AG's) в BFS — это чисто логические структуры, не имеющие каких либо специфичных данных на диске. BFS делит массив блоков, составляющих диск, на куски равного размера, которые называются AG's. Эта сущность используется логикой распределения данных на диске. AG есть просто набор смежных блоков, количество которых привязано к размеру блока и битовой карты.

Рассмотрим 1Gb диск с размером блока ФС в 1kb, т.е. 128 блоков заняты битовыми картами (далее битовой картой мы будем называть 1 bitmap-блок). Размер AG всегда кратен количеству блоков, отображаемых одной битовой картой. Минимальный размер AG составляет здесь 8192 блоков потому, что каждая битовая карта занимает 1024 байт и отображает состояние 8192 блоков. По причинам, обсуждаемым ниже, максимальный размер AG равен 65536 блоков. Выбирая размер AG, BFS учитывает как размер диска, так и разумное количество AG's. Обычно устанавливается 8192 блоков.

BFS старается размещать inode файлов в одной AG с его родительским каталогом, а новые каталоги — в другой AG. Данные файла также помещаются в отличную AG от его inode. Эта политика группирует отдельно метаданные и пользовательские данные файлов, но, естественно, работает только при достаточном количестве дискового пространства.

Битовые карты и координаты каждой AG кэшируются в памяти, для оптимизации так же введен full-флаг, устанавливаемый, если некоторая AG не имеет свободных блоков.

Block-runs (экстенты)

Структура block_run (адекватного перевод этому термину нет, поэтому далее он будет именоваться экстентом, чем он, по сути, и является) есть базовый метод адресации блоков в BFS (см. bfs.h):

```
struct block_run {
      int32       allocation_group;
      uint16      start;
      uint16      length;
      } _PACKED;

typedef block_run inode_addr;
```

Понятно, что allocation_group есть номер AG, start — номер стартового блока экстента (от начала AG), len — его длина. Следовательно, ФС может содержать максимум 2^(32+16) блоков, а 16-битные поля start и len объясняют ограничение размера AG в 65536 блоков. У этой структуры также есть псевдоним — inode_addr: отличие в том, что в inode_addr поле len всегда равно 1.

Суперблок

Суперблок BFS содержит множество полей, несущих информацию о геометрии раздела, геометрии зоны журнала и индексах. Его структура такова (см. bfs.h):

```
struct disk_super_block {
      char        name[BFS_DISK_NAME_LENGTH];
      int32       magic1;
      int32       fs_byte_order;
      uint32      block_size;
      uint32      block_shift;
      off_t       num_blocks;
      off_t       used_blocks;
      int32       inode_size;
      int32       magic2;
      int32       blocks_per_ag;
      int32       ag_shift;
      int32       num_ags;
      int32       flags;
```

```
block_run   log_blocks;
off_t       log_start;
off_t       log_end;
int32       magic3;
inode_addr  root_dir;
inode_addr  indices;
int32       pad[8];
} _PACKED;
```

В ОС Haiku BFS реализована на объектно-ориентированной модели, поэтому все структуры содержат указатели на методы. Здесь они удалены.

Для более надежного определения повреждений данных в суперблоке заложены сразу три magic-номера (соответствующие константы определены в том же файле в enum super_flags).

Первая актуальная информация в суперблоке — размер блока ФС, хранимый в виде прямого количества байт (block_size) и двоичного логарифма этого числа (block_shift). Это сделано для дополнительной проверки целостности файловой системы — в рабочей ФС эти значения согласованы.

Далее идут поля, хранящие размер ФС в блоках (num_blocks) и количество использованных блоков (used_blocks). В BeOS и Haiku эти поля 64-битные, в Linux-версии драйвера они все еще 32-битные (как это было в ранних версиях BFS из-за ограничений компилятора).

Следующее поле — размер inode (inode_size). BFS практикует динамическое выделение inodes, и каждый inode занимает не менее одного дискового блока. Это решение может показаться расточительным, однако оно существенно упрощает схему индексации inodes и вводит дополнительный уровень контроля целостности ФС. Дело в том, что каждый inode хранит этот размер; соответственно, он должен совпадать со значением, хранимым в суперблоке.

AG не имеет ассоциированных дисковых структур данных кроме информации, хранящейся в суперблоке. Поле blocks_per_ag указывает на количество bitmap-блоков, полностью описывающих все блоки AG. Эти bitmap-блоки не должны отображать более 65536 блоков (это верхний предел размера AG). Поле ag_shift хранит двоичный логарифм от blocks_per_ag (также для контроля целостности ФС). Поле num_ags содержит количество AG's на разделе.

Поле flags содержит различные флаги (см. enum super_flags в bfs.h).

Следующая часть суперблока содержит информацию о журнале. Журнал — это массив смежных блоков, в который пишутся вносимые в ФС изменения до их фиксации в файловой системе. Экстент log_blocks описывает всю область журнала целиком, а поля log_start и log_end — начало и конец задействованной в данный момент области (т.к. журнал организован в виде кольцевого буфера).

Последние поля суперблока, root_dir и indices, соединяют суперблок с хранимыми на диске данными. Адрес inode корневого каталога (root_dir) связывает суперблок с корнем иерархии файлов и каталогов раздела, а адрес директории индексов (indices) — с индексами тома (об этом см. ниже).

Структура inode

При работе с файловой системой пользователь применяет понятные ему символьные имена. ФС же удобнее работать с числами, поэму каждому символьному имени соответствует структура inode с уникальным номером. В BFS номером inode является адрес содержащего его блока, т.е., в отличие от файловых систем Linux и UNIX вообще (исключая XFS), номер inode однозначно определяет его местоположение на диске. Это избавляет разработчиков от целого ряда проблем, связанных с организацией поиска inodes при их динамическом выделении.

inode содержит метаданные об ассоциированным с ним объекте файловой системы, будь то файл или каталог. В этой структуре содержится размер файла, данные о его владельце, временные штампы и различные флаги. Наиболее важная часть inode содержит информацию о положении на диске сырых данных файла. См. bfs.h

```
struct bfs_inode {
      int32       magic1;
      inode_addr  inode_num;
      int32       uid;
      int32       gid;
      int32       mode;
      int32       flags;
      bigtime_t   create_time;
      bigtime_t   last_modified_time;
      inode_addr  parent;
      inode_addr  attributes;
      uint32      type;

      int32       inode_size;
      uint32            etc;

      union {
            data_stream       data;
            char              short_symlink[SHORT_SYMLINK_NAME_LENGTH];
```

```
        };
    int32       pad[4];

    small_data  small_data_start[0];
    } _PACKED;
```

Здесь мы снова видим использование magic-номеров для контроля целостности данных. В будущем они смогут использоваться для идентификации различных версий inodes. Для простоты работы с inode в памяти в нем хранится структура inode_addr, указывающая на сам этот inode. Она также может использоваться для контроля целостности данных.

Поля uid/gid содержат информацию о владельце файла. В поле mode хранится информация о разрешениях доступа к файлу, а также идентификатор его типа (регулярный файл, каталог). Модель разграничения доступа к объектам в BFS соответствует спецификации POSIX 1003.1 (традиционная модель для UNIX-систем). Разработчики BeOS планировали реализовать безопасность на основе ACL, однако им не хватило времени. Добавить в BFS ACL достаточно просто и сейчас (при помощи атрибутов), однако разработчики Haiku пока игнорируют эту возможность.

В поле flags хранятся флаги:

```
enum inode_flags {
    INODE_IN_USE            = 0x00000001,
    INODE_ATTR_INODE        = 0x00000004,
    INODE_LOGGED            = 0x00000008,
    INODE_DELETED           = 0x00000010,
    INODE_NOT_READY         = 0x00000020,
    INODE_LONG_SYMLINK      = 0x00000040,

    INODE_PERMANENT_FLAGS   = 0x0000ffff,

    INODE_WAS_WRITTEN       = 0x00020000,
    INODE_NO_TRANSACTION    = 0x00040000,
    INODE_DONT_FREE_SPACE   = 0x00080000,
    INODE_CHKBFS_RUNNING    = 0x00200000,
    };
```

- INODE_IN_USE всегда установлен.
- INODE_ATTR_INODE указывает, что inode ссылается на атрибут.
- INODE_LOGGED. Этот флаг указывает файловой системе, что необходимо журналировать все данные связанного с inode файла. Он используется только для каталогов.
- INODE_DELETED. Этот флаг устанавливается при удалении файла,

когда вызов remove() уже сделан, но ресурсы файла еще не освобождены. Доступ к файлу полностью блокируется. Флаг устанавливается только в in-memory копии inode.

- INODE_NOT_READY. Устанавливается во время инициализации inode. Только в памяти.
- INODE_LONG_SYMLINK. Поток данных содержит символическую ссылку.
- INODE_DONT_FREE_SPACE. Используется chkbfs.
- INODE_CHKBFS_RUNNING. Используется chkbfs.

Смысл остальных флагов будет объяснен далее по мере необходимости.

В отличие от большинства файловых систем UNIX, BFS поддерживает только 2 временных штампа: время создания и последнего изменения. Время последнего доступа исключено для упрощения кода и небольшого повышения производительности. С этим решением нельзя не согласиться, т.к. большинство пользователей UNIX/Linux/BSD монтируют свои ФС с опцией noatime.

Поле parrent, содержащее адрес inode родительского каталога, необходимо для реконструкции полного пути к файлу по номеру inode. Эта возможность используется при обработке запросов (см. далее).

Следующее поле, attributes, возможно, самая условная часть inode BFS. Оно содержит адрес inode, который указывает на каталог, хранящий атрибуты данного файла. Элементы этого каталога есть пары имя/значение для атрибутов. Более подробно устройство этой подсистемы будет рассмотрено в соответствующем разделе.

Поле type используется только в inodes, хранящих атрибуты. Схема индексации атрибутов требует, что бы каждый из них имел обозначенный тип (число, строка, и т.д.). Имя этого поля может сбить с толку — оно не хранит какой-либо информации о типе файла (как это сделано в Apple HFS). BFS хранит хранит данные о MIME-типе файла в атрибуте с именем "BEOS:TYPE".

Поле inode_size используется для проверки целостности inode при загрузке его с диска. Только в самых ранних версиях BFS оно использовалось для других целей.

Поле etc в дисковой копии inode смысла не имеет, в in-memory-экземпляре оно инициализируется указателем на структуру с дополнительными данными о

inode.

Ядро inode — поток данных

Задача inode — связывать файл с некоторым физическим хранилищем. Поле data и есть такая связь. Оно содержит структуру типа data_stream:

```
#define NUM_DIRECT_BLOCKS     12

struct data_stream {
      block_run   direct[NUM_DIRECT_BLOCKS];
      off_t       max_direct_range;
      block_run   indirect;
      off_t       max_indirect_range;
      block_run   double_indirect;
      off_t       max_double_indirect_range;
      off_t       size;
      } _PACKED;
```

Схема адресации блоков соответствует принятой в классических файловых системах UNIX — 12 прямых указателей, одна косвенная и одна двойная косвенная ссылка. Исключения состоит в том, что BFS адресует не отдельные блоки, а экстенты. Т.е. в лучшем случае 12 прямых указателей могут описывать 12*65536=786432 блоков (при 1kb блоке это 768 Mb данных). В худшем случае, на сильно фрагментированной ФС, это всего 12 блоков. На практике прямые указатели адресуют в среднем 100 kb - 10 Mb данных.

Большим файлам 12 прямых указателей не хватает, тогда вступают в действие косвенная (содержащая указатель на блок с прямыми ссылками) и двойная косвенная (указатель на блок с косвенными ссылками) ссылки. В худшем случае максимальный размер файла будет около 1 Gb, в лучшем — приблизительно 34 Gb (для блока в 1 kb).

Атрибуты

Ключевая особенность BFS — это возможность хранить вместе с файлом его атрибуты. Атрибутом называется пара ключ-значение. Эта feature предоставляет файловой системе огромное количество новых возможностей.

В BFS количество атрибутов, ассоциированных с файлом, ограничено только емкостью файловой системы. Имя атрибута всегда строка, значения могут быть интегрированного типа (int32, int64, float, double, строкой любого размера) или raw-данными. BFS может индексировать атрибуты встроенных типов для

эффективного поиска по ним через развитый язык запросов (описан ниже).

Приложения в BeOS используют атрибуты для хранения самых разных данных. Почтовый демон, к примеру, хранит в них различные параметры сообщений, таких, как адрес отправителя, и др. Текстовый редактор хранит в атрибутах стили текста (шрифты, цвета, и т.д.), а сам текст в сыром виде — в потоке данных самого файла. И, конечно, все файлы в ФС имеют атрибут "type".

BFS хранит список ассоциированных с файлом атрибутов в особой директории (на ее inode указывает поле attributes), которая не видна в пространстве имен файловой системы. Файлы в этой директории в поле name содержат имя атрибута, в поле type — тип, а в потоке данных хранят его значение.

Такая схема имеет несколько удобных свойств. Список атрибутов есть просто каталог, а отдельные атрибуты — действительно файлы. Это позволило существенно упростить реализацию подсистемы, однако несколько уменьшило ее производительность (в следствие возникновения необходимости обработки большого количества мелких файлов).

Необходимость в эффективном доступе к разумному количеству маленьких атрибутов была основной причиной, по которой разработчики BFS стали хранить каждый inode в отдельном дисковом блоке. Сам inode занимает немногим более 200 байт, остальное место используется под атрибуты файла. Эта область известна как small_data и используется для хранения тесно упакованного массива атрибутов различного размера (почти 760 байт).

```
struct small_data {
      uint32            type;
      uint16            name_size;
      uint16            data_size;
      char              name[0];
      } _PACKED;
```

BFS помещает первую структуру small_data сразу за структурой inode. Для экономии места структуры не выравниваются. Последняя в блоке структура всегда описывает оставшееся свободное место (конечно, если его не меньше sizeof(struct small_data)).

Все файлы имеют скрытый атрибут, содержащий имя файла (всегда первая small_data). Это сделано для осуществления возможности реконструкции полного пути к файлу по его inode, что необходимо для запросов (см. далее).

Введение структуры small_data существенно усложнило управление атрибутами. Все attribute-операции должны сперва проверять состояния области small_data, а потом переходить к каталогу атрибутов. Несмотря на значительную сложность, это решение позволило существенно выиграть в производительности.

Каталоги

Отображение имен файлов на номера inodes есть основная задача подсистемы каталогов; существует несколько схем такого отображения. Традиционные UNIX-style файловые системы хранят элементы каталогов (пары имя/номер) в виде линейного списка в потоке данных каталога. Этот подход очень прост в реализации, однако не достаточно эффективен при большом количестве файлов. В среднем приходится перебирать до половины каталога для поиска одного конкретного файла.

Более продвинутые файловые системы используют для индексации элементов каталога различные виды бинарных деревьев (обычно B, B+ и B* деревья). Поиск по B-дереву имеет сложность O(log(n)), в то время как поиск по массиву — O(n).

Выбор B+ деревьев для индексации элементов каталога в BFS обусловлен не только желанием улучшить производительность операция с каталогами; более важной причиной послужила необходимость индексирования атрибутов, а использование тех же самых алгоритмов в directory-коде упростило реализацию BFS.

Особенностью подсистемы каталогов BFS является использование строковых имен файлов в качестве ключа в B+дереве (обычно в этом качестве применяется хэш имени — прим. пер.).

Резюме

Структуры, о которых говорилось в этом разделе, претерпели множество изменений в процессе разработки ФС. Они эволюционировали по мере развития проекта, разработчики экспериментировали с различными размерами и алгоритмами.

Структура inode претерпела множество изменений в процессе разработки. В начале inode имел 256 байт в размере, а один блок содержал несколько inodes. Он

также не имел поля small_data для хранения небольших атрибутов, что серьезно било по производительности. Следовательно, управление свободными inodes также становилось проблемой — приходилось вести списки свободных inodes для каждого блока, заголовок глобального списка содержался в суперблоке. Переключение на схему "один блок — один inode" обеспечило место для small_data и упростило управление свободными inodes.

Изначально блок ФС был размером 512 байт, однако у такого решения было 2 недостатка. Такой блок не обеспечивал достаточно места для small_data и плохо согласовывался с алгоритмами B+ дерева (максимальный размер хранимой записи должен быть меньше половины размера узла дерева, т.е. при 255-символьных именах 1024-байтный блок минимален).

Больший размер блока (2 или 4 kb) также неэффективен, т.к. дополнительное место под маленькие атрибуты чаще всего оставалось неиспользованным; B+ деревья, узлы которых должны быть заполнены не менее чем на половину, также, чаще всего, использовали только 1 kb в блоке под полезные данные. (Позволю себе не согласиться с автором книги. Узел B+ дерева каталога в BFS в среднем содержит 10-20 записей по 50-100 байт, что, вообще говоря, неэффективно. Было бы лучше, если бы ключем в таком дереве служило не само имя файла, а его хэш, как это сделано в подавляющем большинстве файловых систем. Тогда достигалась бы куда большая наполняемость узла, и блоки размером 2-4 kb только улучшали бы производительность дерева — прим. перев.).

Концепция групп размещения также задумывалась несколько по-другому. Изначально предполагалось, что каждая AG будет представлять из себя своего рода мини-ФС с отдельными структурами для рапараллеливания выделения/освобождения дискового пространства. Разработчики BFS посчитали, что выгоды от такого решения будут дескридетированы bottleneck'ом в лице централизованного журнала, который все равно не позволит выполнять allocate/free операции действительно параллельно. Это, вообще говоря, не верно. Аналогичная концепция прекрасно работает в XFS и JFS — за счет применения особых схем журналирования глобальных метаданных.

В конечном счете наиболее работоспособной была признана концепция небольших AG's, применяемых, главным образом, для сохранения небольшой размерности адресных структур (block_run) и некоторого упрощения fsck.

Атрибуты и индексирование

Атрибуты

В терминах файловой системы атрибут есть дополнительная информация о файле, которая не хранится в нем самом. В атрибутах могут храниться данные самых разных типов: тип файла, ключевые слова для документа, списки контроля доступа (ACL) и прочее. Важное свойство атрибутов состоит в том, что они позволяют приложениям разделять (share) очень разнородную информацию о файле через простой унифицированный API, избавляя их от необходимости поддерживать собственные базы данных.

API атрибутов

BFS поддерживает следующие операции над атрибутами:

- Запись
- Чтение
- Открытие каталога атрибутов
- Чтение каталога атрибутов
- Обход каталога атрибутов
- Закрытие каталога атрибутов
- Stat на атрибуте
- Удаление атрибута
- Переименование атрибута

Не удивительно, что набор операций над атрибутами подобен набору файловых операций BFS. Для доступа к атрибуту файла приложение должно открыть файл и использовать файловый дескриптор как хэндл для открытия атрибута. Атрибут файла не может иметь выделенного файлового дескриптора и прочитывается поэтому за раз, без открытия и установки позиции (seek). Каталог атрибутов подобен регулярному каталогу — он может быть открыт и пролистан для подсчета количества атрибутов.

Физически атрибут BFS есть пара имя/значение произвольного размера, причем количество атрибутов на файл ограничено только размером диска. BFS должна управлять атрибутами, хранящимися в small_data области и в каталоге атрибутов некоторым унифицированным способом. Каждый раз, когда программа вызывает операцию над атрибутом, BFS ищет его в области small_data, и лишь затем — в каталоге атрибутов. Более тонко обрабатывается случай перезаписи атрибута с изменением его размера, который может потребовать удаления атрибута из small_data и добавления его в attribute-каталог.

В общем случае манипулировать атрибутами в каталоге проще, нежели в small_data, т.к. к подобным объектам применимы обычные файловые операции.

Индексирование

Для понимания индексирования полезно вообразить следующий сценарий: положим, вы приходите в библиотеку и хотите найти книгу. А там, вместо тщательно организованного каталога карточек, вы находите беспорядочную кучу. Каждая карточка содержит полную информацию (атрибуты) о каждой книге. Если карточки не упорядочены, найти нужную вам книгу окажется очень сложно. Т.к. библиотекари предпочитают порядок хаосу, они заводят 3 каталога карточек, организованных в алфавитном порядке и отсортированных по автору, названию и содержанию книги. Это существенно упрощает поиск конкретной книги.

Индексация в файловой системе очень похожа на каталог карточек в библиотеке. Каждый файл в ФС может быть представлен как книга. Если ФС не индексирует информацию о файлах, поиск конкретного файла может привести (в худшем случае) к перебору всех файлов в ФС. При большом количестве файлов поиск может занять очень много времени. Индексация файлов по таким признакам, как имя, размер или время модификации существенно снижает время поиска.

В файловой системе индекс есть простой список файлов, упорядоченных по некоторому критерию. При наличии дополнительных атрибутов вполне естественно проводить индексацию по ним. Т.о., ФС может индексировать e-mail-сообщения по полю "from", а документы по полю "keyword". Индексирование дополнительных атрибутов расширяет функциональность ФС и количество способов, которыми пользователь может упорядочивать информацию.

Благодаря индексированию пользователь может посылать файловой системе сложные запросы типа "Найти все e-mail от Васи Пупкина за последнюю неделю". ФС просматривает свои индексы и выдает список файлов, соответствующих запросу. Конечно, это не значит, что все e-mail-клиенты одинаковы. Производя индексацию в ФС некоторым стандартным способом, мы позволяем приложениям пользоваться функционалом базы данных без собственной реализации таковой.

Хотя файловая система, поддерживающая индексацию, и приобретает черты реляционной базы данных, в полном смысле она таковой не является, т.к. эти системы хранения данных имеют разные задачи. БД несколько проигрывает в

гибкости (обычно имеются записи фиксированного размера, которые сложно расширить после создание БД), однако имеет преимущество в производительности при обработки большого числа элементов, а также более богатый интерфейс запросов. Файловая система предоставляет более общую функциональность и производительность: к примеру, хранение миллионов 128-байтных записей в файловой системе трудно назвать хорошим решением, в то время, как для базы данных это обычное дело.

Упуская многие детали, приведенные примеры дают общее представление о возможностях и задачах индексирования в ФС. Далее мы рассмотрим эти аспекты более подробно.

Что такое индекс?

Первый вопрос, на который следует ответить — что же такое индекс. Индекс есть механизм эффективного поиска по введенным значениям. Используем пример с библиотекой: если мы ищем индекс "автор" со значением "Д. Кнут", мы получим ссылки на книги, написанные этим замечательным программистом. Ссылки позволят нам отыскать физические копии этих книг. Поиск по строковому параметру будет эффективен, т.к. каталог упорядочен по алфавиту.

В ЭВМ индекс есть структура данных, которая хранит пару ключ/значение и допускает эффективный поиск по ключу. Ключи индексов должны быть строго согласованы, т.е. ключ A должен быть строго меньше, больше, или равен ключу B. С обычными типами данных — числовыми или строковыми, обеспечение согласованности не проблема, однако при сравнении данных более сложной структуры ситуация не такая чистая.

В литературе описано множество способов управления и сортировки данных, каждый имеет свои преимущества и недостатки. Файловая система предъявляет ряд требований к механизму индексирования:

- Он должен оперировать с дисковыми структурами
- Он должен иметь приемлемый расход памяти
- Обеспечивать эффективный поиск
- Поддерживать совпадение записей (duplicate keys)

Наиболее общие методы индексирования эффективно работают только в памяти и не применимы в файловой системе. Индексы ФС же должны располагаться на постоянном хранилище и переживать перезагрузки и крэши.

Далее, т.к. файловая система это всего лишь часть ОС, использование индексов не должно существенно сказываться на остальной системе, т.е. расходовать слишком много памяти или процессорного времени. Следовательно, в память не может быть загружен ни весь индекс, ни сколь-нибудь значительная его часть. Т.к. индексов может быть много, ФС должна иметь возможность загружать любое требуемое их количество и легко переключаться между ними, не расходуя лишнее процессорное время. Эти условия исключают из рассмотрения несколько техник индексирования, часто применяемых в коммерческих БД.

Основное требование к механизму индексирования — эффективность поиска по ключу. Этот параметр может иметь решающее значение для производительности все файловой системы, т.к. каждый доступ к имени файла сам по себе требует поиска. Поэтому поиск файла в ФС должен быть наиболее эффективен по его индексу.

Последнее требование, и, возможно, самое сложное — необходимость поддержки совпадающих ключей. Необходимость этой, на первый взгляд, необязательной возможности видна на примере одинаково поименованных файлов, расположенных в разных каталогах.

Выбор структур данных

Из всего множества существующих структур данных, применяемых для индексирования, для использования в файловых системах пригодны лишь немногие. Наиболее популярные структуры данных для хранения дисковых индексов это B-деревья в различных своих вариантах (B*, B+деревья); также хэш-таблицы могут быть расширены для применения на дисковых структурах. Каждый подход имеет свои преимущества и недостатки. Ниже мы кратко рассмотрим различные подходы и их возможности.

B-деревья

B-деревья — древовидные структуры, организующие данные в совокупность узлов. Подобно настоящим деревьям, B-деревья имеют корневой — стартовый — узел. Он имеет связи с нижележащими узлам, которые, в свою очередь, также связаны с более низкими уровнями — пока не будет достигнут листовой уровень. На листовом уровне расположены узлы, не имеющие нисходящих связей.

Ключом к производительности B-деревьев является их постоянная сбалансированность — все ветви в дереве имеют одинаковую длину в любой

момент времени.

Вставка в дерево начинается с поиска желаемой позиции (через стандартную операцию поиска в дереве). Если вставка ключа может вызвать переполнение узла, происходит операция разделения узла на 2 полу-заполненных. При этом в родительский узел вставляется новый указатель. В худшем случае (если заполнение подходящих узлов дерева близко к единице) эта операция может быть распространена вверх на множество узлов, включая корневой (т.е. вызвать создание нового корня).

Удаление ключа из узла протекает подобно вставке, только вместо разделения узлов оно может вызвать объединение. Слияние соседних узлов приводит к модификации родителя с удалением одного указателя. Детально с процессами балансировки дерева вы можете ознакомится в [3].

Концепция B-дерева разрабатывалась специально для организации данных на медленном постоянном хранилище (таком, как жесткий диск). Каждый узел дерева имеет фиксированный размер, обычно совпадающий с размером блока файловой системы. B-дерево легко хранить в одном файле, тогда межузловые связи становятся просто смещениями в пределах файла. Хотя может показаться, что операция балансировки дерева требовательна к вычислительным ресурсам системы — это не так, ведь перемещения данных при этом не происходит.

Хэширование

Хэширование — альтернативная техника упорядочивания данных на диске. При хэшировании исходный ключ проходит через функцию, генерирующую для него т.н. числовой хэш по некоторому алгоритму. Хэш используется в качестве индекса в таблице, содержащей пары ключ/значение. Основное преимущество хэширования — постоянство времени поиска в хэш-таблице.

Естественно, разные входные ключи могут генерировать совпадающие хэши — коллизии. Один из методов преодоления этой ситуации — компоновка связанного списка из совпадающих хэшей — т.н. цепочки. Другой (достаточно экзотический) метод — повторное хэширование входных данных другой функцией — вплоть до получения уникального хэша.

Еще одна проблема хэширования состоит в том, что с ростом размера хэш-таблицы растет и количество коллизий, т.е. падает эффективность поиска по таблице.

Усовершенствованный подход — т.н. расширенное хэширование — разбивает таблицу на 2 части. Применяется файл, содержащий каталог указателей на пакеты данных, и файл с самими пакетами. В качестве ключа в каталоге пакетов используются несколько бит значения хэша. При переполнении пакета принимается решение об увеличении количества этих бит — что приводит к достаточно расточительной операции перестроения таблицы.

Файловая система предъявляет несколько достаточно жестких требований к алгоритмам упорядочивания данных, таких, как умеренный побочный расход дискового пространства и хорошая масштабируемость. Очевидно, что хэш-таблицы, хотя и расходуют диск достаточно экономны, совершенно не масштабируемы — т.е. теряют эффективность с ростом объема индексируемых данных. С некоторыми частными оптимизациями расширенное хэширование все же может применяться в файловых системах (и с успехом применяется в JFS и ZFS — прим. пер.), однако B-деревья гораздо лучше соответствуют поставленным задачам (с чем тоже нельзя не согласится — прим. перев.) — и именно поэтому были выбраны для BFS.

Связь: индексы и файловая система

Самые очевидные вопросы, которые необходимо задать в этом месте — как управляется список индексов? Где "живут" отдельные индексы? То есть в каком именно месте стандартной структуры файлов и каталогов они помещаются? BFS использует нормальную структуру каталогов для управления списком индексов. ФС хранит данные каждого индекса в регулярном файле, помещающемся в в специальной директории индексов.

Хотя нет проблемы разместить файлы индексов в обычной, т.е. видимой пользователю директории, BFS хранит эти файлы в скрытом каталоге, создаваемом одновременно с файловой системой. Суперблок хранит номер inode для этого каталога, который и устанавливает связь с остальной файловой системой. Такой ход предотвращает случайное удаление файлов индексов или другие ошибки пользователя и приложений, которые могут оказать катастрофическое воздействие на ФС. Единственный недостаток этого хода — необходимость введения специального API для доступа к индексам.

Программный интерфейс работы с индексами очень прост и содержит реализацию следующих операций:

- создание

- удаление
- открытие каталога индексов
- чтение этого каталога
- stat

BFS не поддерживает более продвинутые операции над индексами, т.к. в этом практически нет нужды. Метод create принимает в качестве параметров имя индекса и тип данных. Имя индекса связывает его с соответствующим атрибутом, который в последствии будет использоваться данным индексом. Например, почтовый демон BeOS добавляет атрибут с именем "MAIL:from" для всех получаемых e-mail сообщений, а также создает индекс с именем именем "MAIL:from" Тип данных индекса должен соответствовать типу данных атрибута. BFS поддерживает следующие типы данных:

- Строка (до 255 байт)
- 32-битный целый
- 64-битный целый
- плавающий одинарной точности (float)
- плавающий двойной точности (double)

Другие типы также возможны, но представленные обеспечивают наиболее общую функциональность. На практике большинство индексов это строки.

Одна из проблем при создании индекса состоит в том, что его имя может совпасть с именем уже существующего атрибута. Например, если файл имеет атрибут "foo", и программа создает индекс с таким же именем, существующий файл в этот индекс не добавляется. Сложность в том, что не существует простого способа определить, какие файлы уже имеют данный атрибут без обхода всех файлов в ФС. При необходимости приложение может сделать это самостоятельно, ФС не заботится об этом.

Удаление индекса есть простая операция, состоящая в стандартном удалении соответствующего файла из индексной директории. Однако удалять индексы, вообще говоря, не рекомендуется, т.к. их пересоздание не приведет к переиндексации файлов, отмеченных данным атрибутом. Поэтому индекс должен удаляться только при удалении из системы всех приложений и файлов, его использующих (т.е. когда он пуст).

Оставшиеся функции index-API служебные. Метод stat позволяет приложению проверять существование индекса и получать его размер. Эти

функции имеют тривиальную реализацию, т.к. все используемые структуры данных идентичны применяемым в работе с регулярными файлами и каталогами.

Автоматическое индексирование

Кроме предоставления пользователям возможности создавать собственные индексы, BFS поддерживает индексы по встроенным атрибутам файлов: имени, размеру и времени последней модификации. Файловая система сама полностью поддерживает эти индексы, т.к. имя, размер и mtime не являются регулярными атрибутами файла, а содержаться в inode и не управляются кодом подсистемы атрибутов.

Индекс по имени содержит список имен файлов всей ФС. Каждый раз при изменении имени файла (создании, удалении и переименовании), файловая система обновляет индекс. Процедура обновления индекса имени является частью транзакции создания файла и фиксируется только вместе с оной. Если имя файла не может быть добавлено в индекс по причине нехватки места — файл не создается.

Удаление индекса несколько менее проблематично, т.к. оно не может провалиться из-за нехватки места. Эта процедура является завершающей частью транзакции удаления файла.

Операция переименования — одна из сложнейших функций, как вообще, так и касательно индексов. Обновление индекса есть завершающая стадия rename-транзакции. Сам метод rename() включает удаление оригинального имени (если оно существует) и вставку нового имени в индекс. Откат транзакции по причине провала вставки очень проблематичен. Операция rename() может получить в середине своего выполнения удаленный файл, если новое имя уже существует. Однако, т.к. файл удален (и все его ресурсы уже освобождены), откат такой транзакции чрезвычайно сложен. Из-за сложности и редкости возникновения таких ситуаций (только при стопроцентном заполнении диска) BFS даже не пытается обработать этот случай. Раздел по-прежнему остается в согласованном состоянии, просто такой файл не будет проиндексирован.

Обновление size-индекса происходит при изменении размера файла. В целях оптимизации ФС обновляет size-индекс в момент закрытия файла (что бы не делать этого при каждой записи в файл). Минус в том, что для открытых в данный момент файлов индекс может оказаться несколько устаревшим, однако выигрыш а производительности от этого решения очень значительный.

Другая проблема size-индекса проявляется при наличии большого количества файлов одинакового размера. На практике это очень редкая ситуация, однако она часто возникает в момент проведения тестов производительности (benchmarks), когда в системе создается множество файлов — зачастую одного размера. Производительность ФС в этом случае убывает нелинейно, отставание становится заметным при 10000 совпадающих ключей.

mtime-индекс обновляется также при закрытии файла: удаляется индекс со старым временем и вставляется с новым. Зная, что встроенный mtime-индекс оказывает решающее воздействие на производительность файловой системы, BFS использует сложную технику для повышения его эффективности. Время последней модификации в inode имеет шаг в 1 секунду, за которую могут быть созданы тысячи файлов, BFS расширяет значение mtime в индексе с 32 до 64 бит и добавляет небольшую случайную компоненту для уменьшения количества коллизий.

Кроме этих встроенных атрибутов в различных версиях BFS могут индексироваться и другие. В ранних релизах существовал индекс по времени создания файла, удаленный из-за практической невостребованности и потерь производительности (удаление этого и индекса позволило на 20% повысить производительность массового создания и удаления файлов).

Прикладные индексы

Наряду со встроенными системными индексами существуют и прикладные, создаваемые программами. Вспомним пример с почтовым демоном. При первом запуске почтовая система создает индексы по всем требуемым ей атрибутам. Когда программа записывает один из этих атрибутов в файл, BFS отмечает, что существует соответствующий индекс и обновляет его согласно значению атрибута файла.

При каждой записи атрибута файловая система должна просматривать индексный каталог на предмет наличия соответствующего индекса. На первый взгляд может показаться, что это снижает скорость обработки простых, неиндексируемых атрибутов. Однако с учетом того, что индексов в системе, как правило, не более 100 и все они кэшированы в памяти, этот поиск проходит очень быстро.

B+деревья BFS

BFS использует B+деревья для хранения директорий и всей индексной информации. Код BFS обеспечивают хранение в дереве ключей переменного размера вместе с единственным дисковым смещением (64 бита в BFS), и обработку совпадения ключей — коллизий. Ключи могут быть строкового типа, integer (32 и 64 бита), float и double.

API

Интерфейс B+ дерева достаточно просто и содержит 6 основных методов:

- открытие/создание дерева
- вставка пары ключ/значение
- удаление пары ключ/значение
- поиск ключа с возвратом его значения
- переход в начало/конец дерева
- линейный обход листьев дерева (прямой/обратный)

Процедура создания дерева принимает несколько параметров: размер узла, тип сортируемых данных и несколько флагов, определяющих внутреннюю логику. Все деревья BFS оперируют 1-kb узлами — независимо от размера блока ФС. Это число было выведено экспериментально и оптимизировано для индексации каталогов: BFS поддерживает имена файлов длиной до 255 байт, следовательно, 512-байтные узлы были бы слишком малы, а при большем размере узла в большинстве каталогов не достигалось бы 100% заполнение (Мотивом этого решения, очевидно, является оптимизация подсистемы каталогов для индексации атрибутов — файлов с небольшими (6-12 символов) именами — в надежде сэкономит время на подсчете хэшей. Однако механизм сравнения строк однозначно медленнее сравнения двух хэшей, поэтому выгода от такого решения выглядит сомнительной — прим. пер.).

Процедура вставки принимает в качестве параметров ключ (соответствующего дереву типа), его длину и собственно данные. Данные — это всегда 64-битный номер inode. При возникновении коллизии в дереве, не допускающем совпадения ключей, возвращается ошибка, иначе ключ вставляется рядом с дубликатом.

Процедура удаления принимает пару ключ/значение и производит поиск в дереве по ключу. Если находятся несколько дубликатов ключа — сравниваются значения.

Процедура поиска принимает ключ и возвращает первый попавшийся дубикат.

Структуры данных

Структура B+дерева BFS достаточно проста и прозрачна (вспомним, для сравнения, деревья JFS или XFS! — пер.). На диске дерево — это совокупность узлов. Самый первый узел дерева — своего рода "суперблок" — содержит заголовок, описывающей дерево (см. BplusTree.h):

```
struct bplustree_header {
      uint32      magic
      uint32      node_size;              /* размер узла */
      uint32      max_number_of_levels;   /* текущая высота дерева */
      uint32      data_type;              /* тип индексируемых
                                           * данных, см. enum
                                           * bplustree_types */
      off_t       root_node_pointer;      /* адрес корневого узла */
      off_t       free_node_pointer;      /* адрес первого
                                           * свободного узла */
      off_t       maximum_size;           /* текущий размер файла дерева
                                           * (вместе со свободными
                                           * узлами) */
} _PACKED;
```

Поле free_node_pointer содержит голову связанного списка свободных узлов. Каждый такой узел содержит адрес (смещение в пределах файла) следующего свободного узла или -1 в случае достижения конца списка.

Листовые и внутренние узлы дерева имеют одинаковую структуру и состоят из заголовка, массива ключей, массива длин ключей и массива значений. Листовые узлы содержат пользовательские данные, внутренние — указатели на потомков. Отличаются по заголовку.

```
struct bplustree_node {
      off_t       left_link;        /* ссылки на левого и правого
                                     * соседей, только для листьев */
      off_t       right_link;
      off_t       overflow_link;    /* для внутренних узлов */
      uint16      all_key_count;    /* количество ключей */
      uint16      all_key_length;   /* суммарная длина всех ключей */

      /* массив ключей */

      /* массив длин ключей – хранит конечные индексы для
       * каждого ключа  */
```

```
        /* массив значений */
} _PACKED;
```

Мнение

Знакомство с BFS производит странное впечатление — эта ФС сочетает в себе как достаточно современные (выделение блоков экстентами, B+деревья в каталогах), так и уже "архаичные" (битовые карты, косвенные блоки) черты. Файловая система хорошо оптимизирована для работы с большим количеством мелких, редко модифицируемых файлов — поиск по хранилищу документов или музыки, за счет применения индексов, будет экстремально эффективным. Благодаря достаточно грамотной политике размещения метаданных (inodes, каталоги и данные файлов — в разных AG) BFS также довольно эффективно противодействует фрагментации.

Из минусов: возможности групп размещения к распараллеливанию операций выделения/освобождения блоков (как в XFS и JFS) практически не используются; вся файловая система ориентирована на обработку индексов и атрибутов, адаптивная оптимизация (по типу данных пользователя) отсутствует. На статичном хранилище это оказывается очень полезно, однако на активно используемой файловой системе с разнородными данными производительность BFS оказывается значительно ниже, чем у аналогов.

Источники

1. Dominic Giampaolo "Practical File System Design with Be File System".
2. Исходники Haiku от 20.09.2007.
3. Д. Кнут "Искусство программирования, том 3".