САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет
Кафедра системного программирования

Серко Сергей Анатольевич

Организация эффективного хранения образов виртуальных машин с возможностью их модификации для автозапуска приложений

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н. профессор Терехов А.Н.

_____
подпись

Научный руководитель:
ст. преп. Баклановский М.В.

_____
подпись

Рецензент:
первый зам. Ген. директора – зам. Ген. конструктора
ОАО "ВПК "НПО машиностроения" по ИТ
Мартынов В.И.

_____
подпись

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics&Mechanics Faculty
Software Engineering Chair

Serko Sergey

Organising efficient storing of virtual machine images, modifiable for application autorun

Graduation Thesis

Admitted for defence.
Head of the chair:
Professor Andrey Terekhov

_____
signature

Scientific supervisor:
Senior Lect. Maxim Baklanovsky

_____
signature

Reviewer:
First Deputy Director General, Deputy Designer General of
JSC NPO Mashinostroyenia on IT
Vyacheslav Martynov

_____
signature

Saint-Petersburg
2014

Оглавление

Введение 5
Постановка задачи 6
Глава 1. Организация эффективного хранения образов виртуальных машин с использованием методов хранения/сжатия информации на основе архивации и дедупликации 7
1.1 Основные алгоритмы архивации 7
1.1.1 RLE 8
1.1.2 Кодирование Хаффмана 8
1.1.3 Арифметическое кодирование 9
1.1.4 MTF 10
1.1.5 BWT 11
1.1.6 PPM 13
1.1.7 LZ77 14
1.1.8 LZSS 15
1.1.9 LZMA 16
1.1.10 LZH, LZARI 17
1.1.11 Deflate 17
1.2 Тестирование решений на основе архивации 18
1.3 Решения на основе дедупликации 21
1.3.1 SDFS 22
1.3.2 LessFS 23
1.3.3 ZFS on Linux 23
1.4 Тестирование решений на основе дедупликации 24
Глава 2. Разработка спецификации файловой системы специального назначения 26
2.1 Fork измененного файла 29
2.2 Основные операции 29
2.2.1 Добавление файла 29
2.2.2 Удаление файла 29
2.2.3 Fork 29
2.2.4 Изменение файла 30
2.2.5 Чтение файла 30

Глава 3. Статическая модификация образов виртуальных машин для внедрения автозапуска приложений 31
3.1 Автозапуск приложений 31
3.1.1 Автозагрузка приложений в ОС семейства Windows 31
3.1.2 Автозагрузка приложений в ОС семейства Linux 32
3.1.3 Автозагрузка приложений в Mac OS 33
3.2 Внедрение автозапуска приложений в ВМ 35
3.3 Работа с внешними файлами 35
Глава 4. Разработка прототипа модуля эффективного хранения образов виртуальных машин с возможностью их модификации для автозапуска приложений 37
4.1 Базовая функциональность 37
4.2 Технологии 37
4.3 Архитектура 37
Заключение 39
Дальнейшее развитие 39
Список литературы 40
Приложение 1. Спецификация Fork-FS 42
Приложение 2. Результаты тестирования архиваторов. Архивация образа windows_office.vmdk. 72
Приложение 3. Результаты тестирования средств дедупликации. 74

Введение

Одна из актуальных проблем в сфере информационных технологий – обеспечение безопасности информационных систем, в частности, организация одновременной работы с данными разных уровней секретности. При работе с данными разных уровней конфиденциальности особое внимание следует уделять угрозам, которые могут привести к рассекречиванию информации. Основным источником таких угроз являются вредоносные программы, распространяемые через Интернет: во время обработки секретных данных пользователь может перейти по вредоносной ссылке и тем самым поставить под угрозу их конфиденциальность.

Наиболее очевидный способ решения этой проблемы заключается в разделение работы с данными различных уровней секретности между физически разными машинами. Однако, этот метод обладает целым рядом недостатков: пользователю неудобно работать одновременно за несколькими компьютерами, предприятию приходится повышать затраты на закупку оборудования и его обслуживание.

Задачу безопасной работы с данными различных уровней секретности на одной машине призван решить проект MCD, инициированный на кафедре СП Мат-Мех ф-та СПбГУ. В основе данной системы лежат технологии виртуализации и удаленной доставки приложений. Безопасность рабочего пространства пользователя достигается путем исполнения требуемых пользователю приложений в отдельных виртуальных машинах на безопасном сервере и удаленной работой с ними посредством протокола X11. На пользовательскую машину с тонким клиентом, не имеющую сторонних программ и выхода в Интернет, транслируется лишь видео-поток, состоящий из мгновенных снимков приложения. Пользователь видит программу и взаимодействует с ней так, как если бы программа работала локально, только все управляющие действия – движение курсором, нажатия кнопок мыши и клавиш клавиатуры – отправляются на сервер. Там элементы взаимодействия обрабатываются, и обновленное состояние приложения передается пользователю.

Как уже было сказано выше, в системе MCD запуск приложения означает запуск отдельной виртуальной машины. Потенциально большое количество одновременно работающих пользователей и доступных им приложений ставят вопрос об организации эффективного хранения порождаемых образов виртуальных машин, занимающих значительные объемы дискового пространства. Одним из решений этой проблемы является использование систем хранения данных с большим количеством дисковых накопителей. Однако данный вариант подходит далеко не всем ввиду его дороговизны.

Как следствие встает задача оптимального использования доступного дискового пространства.

Сегодня лидирующими технологиями в сфере сжатия данных являются дедупликация и архивация. Они существенно уменьшают объем используемого образами виртуальных машин дискового пространства, но за это приходится расплачиваться дополнительными временными затратами на распаковку/восстановление оригинального файла. Специфика системы MCD накладывает ограничение на время доступа к файлу – не каждый пользователь готов ожидать до 5 минут для запуска приложения, пусть и в безопасной среде. Исходя из вышесказанного, можно сформулировать задачу организации эффективного хранения образов виртуальных машин, где эффективность понимается как совокупность оптимального использования дискового пространства и высокой скорости доступа к информации.

В данной дипломной работе исследованы такие технологии сжатия информации как дедупликация и архивация, проведены тестирование и сравнительный анализ решений на основе этих технологий; доказана возможность построения системы эффективного хранения образов виртуальных машин, а также описано построение прототипа системы эффективного хранения образов виртуальных машин, обладающей функцией внедрения автозапуска приложений.

Постановка задачи

Целью работы является организация эффективного хранения виртуальных машин с возможностью их модификации для автозапуска приложений в системе Multi-Cloud Desktop. Для достижения цели были выделены следующие задачи:

1. Изучение подходов хранения/сжатия информации;
2. Тестирование и сравнительный анализ существующих решений на основе технологий архивации и дедупликации, тестирование и анализ комплексных решений, реализованных на основе существующих;
3. Поиск или разработка собственного метода хранения/сжатия образов виртуальных машин, отвечающего требованиям конкурентоспособной производительности и эффективного использования дискового пространства;
4. Внедрение автозапуска приложений в образы ВМ;
5. Разработка прототипа модуля эффективного хранения образов виртуальных машин с функцией внедрения автозапуска приложений.

Глава 1. Организация эффективного хранения образов виртуальных машин с использованием методов хранения/сжатия информации на основе архивации и дедупликации

Наиболее распространёнными технологиями сжатия информации на сегодняшний день являются дедупликация и архивация. На их основе реализовано большое количество программных решений. Ввиду разнообразия алгоритмов сжатия информации и основанных на них программных продуктов, чтобы найти оптимальное решение для модуля хранения в системе MCD, имеет смысл протестировать их скоростные и качественные характеристики на задаче хранения образов виртуальных машин. Выбор рассматриваемых программных средств, исходя из возможности их использования в проекте, был ограничен бесплатными приложениями, обладающими интерфейсом командной строки.

1.1 Основные алгоритмы архивации

Архивация – алгоритмическое преобразование данных, производимое с целью уменьшения занимаемого ими объёма. Существует значительное количество алгоритмов архивации, их все можно разделить на две большие группы – неискажающие и искажающие.

Искажающие алгоритмы сжимают информацию с необратимыми потерями, например, за счет удаления несущественной части данных. После компрессии одним из таких методов полное восстановление невозможно. Зачастую искажающие алгоритмы применяют для сжатия видео, аудио, изображений, где удаление избыточной информации означает незначительное искажение картинки или звука, которые не сильно мешают нормальному восприятию исходных данных. Среди искажающих алгоритмов можно выделить группу специальных методов сжатия, которые оперируют информацией о природе данных, за счет чего повышается скорость и/или степень сжатия.

Неискажающие алгоритмы сжатия информации, в свою очередь, гарантируют, что данные, полученные в результате декомпрессии, будут в точности совпадать с исходными. Алгоритмы именно этой группы применимы для сжатия бинарных файлов, в том числе образов виртуальных машин. Неискажающие алгоритмы могут быть разбиты на 2 подгруппы: алгоритмы статистического (контекстного) моделирования и алгоритмы словарного сжатия. Суть алгоритмов статистического моделирования заключается в том, что процесс сжатия информации разбит на два этапа – построение модели данных и их кодирование на основании результатов моделирования. Идея алгоритмов на основе

словарного сжатия заключается в замене строк на коды, соответствующие их индексам в некотором словаре.

Ниже будут рассмотрены основные алгоритмы сжатия без потерь и некоторые вспомогательные алгоритмы преобразования данных, используемые в популярных современных архиваторах.

1.1.1 RLE

Кодирование длин повторов (Run-length encoding) – один из наиболее старых и простых алгоритмов компрессии. Сжатие происходит за счет замены цепочек одинаковых символов исходной строки на пары "счетчик, значение". Обычно, в качестве символов выступают байты исходного сообщения.

Одна из реализаций алгоритма кодирования длин повторов состоит из таких этапов:

1. Поиск наименее часто встречающегося байта, который будет выступать в роли префикса;
2. Замена цепочек повторяющихся символов на тройки <префикс; счетчик; значение>. Если в исходном сообщении наименее встречающийся байт встречается несколько раз подряд, то он заменяется на пару <префикс; счетчик>.
3. В качестве символа конца файла используется пара <префикс; 0>.

Обратное преобразование осуществляется очевидным образом: каждый символ из декодируемой последовательности дублируется указанное количество раз.

Этот алгоритм используется для компрессии изображений в форматах PCX, TIFF, BMP.

1.1.2 Кодирование Хаффмана

Кодирование Хаффмана – метод сжатия информации, основанный на двоичных кодирующих деревьях и вероятностях вхождения символов в кодируемое сообщение, который был предложен Д.А.Хаффманом в 1952 году. Идея алгоритма заключается в построении кодов переменной длины, состоящих из целого числа битов, в зависимости от вероятности вхождения символов. Сжатие информации происходит за счет того, что символам с большей вероятностью присваиваются коды меньшей длины. Алгоритмы, использующие схожий принцип сжатия, заключающийся в замене символов их кодовыми

словами – строками нулей и единиц – так, что более часто встречающимся символам соответствуют более короткие слова, называются методами энтропийного кодирования.

Алгоритм сжатия по методу Хаффмана состоит из следующих шагов:

1. Входной алфавит образует список свободных узлов дерева, причем каждый лист имеет вес равный либо вероятности, либо количеству вхождений символа в сообщение;
2. Выбирается 2 символа с наименьшими весами;
3. Создается родитель с весом равным их суммарному весу;
4. Родитель добавляется в список свободных узлов, его потомки удаляются из списка;
5. Одной дуге, выходящей из родителя сопоставляется 0, другой – 1;
6. Повторить 2-5, пока не останется 1 узел в списке свободных – он будет корнем дерева;
7. Осуществляется замена символов исходного сообщения полученными кодами.

Код символа определяется как путь от листа до корня. Поскольку в результате работы этого алгоритма ни один из получаемых кодов не является префиксом другого, они могут быть однозначно декодированы, несмотря на их переменную длину. Существенным недостатком этого алгоритма является то, что для восстановления исходных данных декодеру необходимо располагать таблицей частот, которая была использована при кодировании. Это приводит к увеличению размера сжатого файла. Еще одним недостатком является необходимость предварительного анализа сообщения для построения таблицы частот.

Коды Хаффмана используются на некоторых этапах создания архивов форматов ZIP и GZIP, а также при сохранении изображений в формате JPEG.

1.1.3 Арифметическое кодирование

Алгоритм Хаффмана эффективен в случае, когда частоты появления символов пропорциональны $1/2^n$, где n – натуральное, поскольку коды Хаффмана для каждого символа состоят из целого числа бит. Если частоты появления символов не подходят под вышеописанное правило, то это приводит к удлинению префиксного кода Хаффмана, а значит к ухудшению сжатия данных.

Решением этой проблемы является арифметическое кодирование, основной идей которого является присваивание кодов не отдельным символа, а их последовательностям.

Арифметическое кодирование, как и алгоритм Хаффмана, является методом энтропийного кодирования, а значит, использует для сжатия информацию о частоте использования каждого символа алфавита. Основным рабочим элементом метода арифметического кодирования является рабочий отрезок = полуинтервал [a, b) с точками, расположенными в нем таким образом, что длины образованных ими отрезков равны частоте использования символов; каждый такой отрезок соответствует определенному символу с данной частотой появления.

На первом шаге алгоритма в качестве рабочего отрезка используется [0,1). На нем вышеописанным способом располагается точки. Основной шаг алгоритма заключается в том, что для очередного кодируемого символа ищется соответствующий отрезок, и назначается рабочим, при этом происходит его разбиение точками аналогично проведенному на первом шаге алгоритма. Основной шаг повторяется для некоторого количества символов кодируемого сообщения. Результатом кодирования цепочки является пара <любое число из итогового рабочего отрезка; количество закодированных символов>, либо просто любое числом из рабочего отрезка, но во втором надо ограничивать каждую кодированную цепочку некоторым уникальным символом, чтобы декодер мог точно определить конец закодированного сообщения. Надо заметить, что зачастую для простоты кодирования выбирается не любое число из интервала, а его нижняя граница.

Для работы декодирующего алгоритма помимо декодируемого сообщения необходимо иметь значения частот символов. Декодер работает следующим образом: на каждом шаге алгоритма проверяется, в каком полуинтервале рабочего отрезка лежит код – очередным символом декодируемого сообщения является символ, соответствующий этому полуинтервалу, который назначается рабочим отрезком для следующего шага алгоритма. Итерации продолжаются до тех пор, пока не будет встречен код конца сообщения, либо, в случае кодирования строк парами <число из рабочего отрезка, количество символов>, пока не будет совершено заранее известное количество шагов алгоритма.

1.1.4 MTF

MTF (Move-To-Front) – это вспомогательное преобразование данных, целью которого является улучшение производительности последующего кодирования. Данный алгоритм был опубликован в 1980 году под названием «стопка книг», поскольку последовательность его действий напоминает перекладывание наиболее популярных книг из отдаленных стеллажей библиотеки ближе к рабочему месту библиотекаря.

Основная идея «движения вверх» заключается в замене каждого символа кодируемой строки его номером в алфавите, причем, чем ближе этот символ к началу алфавита, тем короче его код. После замены очередного символа его кодом производится трансформация алфавита – замененный символ помещается в начало алфавита, сдвигая остальные все остальные литеры в нем на одну позицию. Таким образом, наиболее используемые символы будут кодироваться меньшими значениями, а последовательность одинаковых символов, начиная со второго, будет заменена на последовательность нулей. Из вышесказанного следует, что насыщенная длинными повторами разных символов входная строка будет преобразована в блок данных, самыми частыми символами которого будут нули, что значительно повышает эффективность последующего энтропийного кодирования.

Для алгоритма обратного преобразования необходимо располагать исходным алфавитом. Декодирование происходит очень похожим на кодирование способом: элементы входной строки являются номерами символов из алфавита; по мере алфавитной замены номеров на символы исходной последовательности, заменяющий на очередном шаге алгоритма символ перемещается в начало алфавита со сдвигом остальных литер на одну позицию.

Move-to-front является одним из шагов алгоритма Deflate и используется в архиваторе bzip.

1.1.5 BWT

BWT (Burrows-Wheeler transform) – вспомогательный алгоритм преобразования данных, описанный в 1994 году исследователями Майклом Барроузом и Дэвидом Уилером (Michael Burrows & David Wheeler), который используется для представления данных в более легко сжимаемом виде. Хоть данный алгоритм сам по себе не занимается компрессией данных, тем не менее, исторически называется блочно-сортирующим сжатием. Как видно из этого названия, BWT оперирует целым блоком данных. Это значит, что для работы алгоритма, ему должны быть заранее известны все элементы входного потока или достаточно большой его части, что делает BWT непригодным для сжатия «на лету» – символ за символом.

Алгоритм работы BWT состоит из следующих этапов:

1. Выделение блока данных;
2. Создание матрицы циклических перестановок символов из выделенного блока;

3. Сортировка матрицы в лексикографическом порядке, с запоминанием номера исходной строки.

Последний столбец полученной в результате описанных преобразований матрицы является результатом работы BTW. Однако для декодирования требуется дополнительная информация, а именно номер строки матрицы, содержащий исходную последовательность символов, который кодировщик запомнил на 3ем шаге алгоритма.

Процесс декодирования описывается следующим алгоритмом:

1. Символы декодируемого сообщения образуют столбец, к которому слева приписывается колонка, образованная в результате упорядочивания символов декодируемого сообщения в лексикографическом порядке, формируя первый столбец сортированной матрицы циклических перестановок исходных данных;
2. Строки полученной матрицы нумеруются от 0 до N-1;
3. Матрица сортируется в лексикографическом порядке по последнему столбцу, в результате чего образуется некоторая перестановка нумерации строк. Полученная перестановка называется вектором обратного преобразования;
4. Берется элемент вектора обратного преобразования, соответствующий номеру исходной строки в матрице циклических перестановок. Он является индексом первого символа исходного сообщения в декодируемой строке;
5. Производится итерация по элементам вектора обратного преобразования: на каждом следующем шаге берется элемент этого вектора, соответствующий индексу, полученному на предыдущем шаге алгоритма. На первом шаге итерации выбирается элемент, соответствующий индексу, полученному в п. 4. Выбранный элемент будет являться индексом следующего символа исходного сообщения в декодируемой строке.

Как было сказано выше, данные, полученные в результате преобразования Барроуза-Уилера, обладают рядом свойств, которые позволяют значительно упростить процесс их сжатия:

1. Порядок символов во входной строке меняется таким образом, что повторяющиеся подстроки образуют на выходе идущие подряд последовательности одинаковых символов.
2. Подстроки, имеющие незначительные отличия, дают на выходе последовательности одинаковых символов, изредка перемежающиеся другими символами.

Первое свойство позволяет использовать BWT вместе с алгоритмом RLE для задачи сжатия исключением повторяющихся подстрок. Второе свойство означает, что в случае применения к преобразованным данным алгоритма MTF, полученный набор чисел будет иметь очень хорошее распределение для применения кодирования Хаффмана или любого другого энтропийного сжатия.

1.1.6 PPM

Как говорилось выше, процесс сжатия на основе алгоритмов статистического моделирования разбит на построение модели данных и их кодирование на основании построенной модели. Одним из методов статистического моделирования является PPM (Prediction by Partial Matching — предсказание по частичному совпадению), предложенный в 1984 г. Джоном Клири и Ианом Виттеном (John Cleary & Ian Witten). Данный алгоритм основан на вероятностной оценке появления символа, в зависимости от его контекста. Контекстом символа называется непосредственно предшествующая ему строка определенной длины. Когда для оценки вероятности символа используется его контекст длины N, говорят об использовании модели PPM порядка N.

Использование модели порядка 0 эквивалентно контекстно-свободному моделированию, когда вероятность символа определяется исключительно из частоты его появления в сжимаемом потоке данных. Зачастую эта модель используется вместе с кодированием Хаффмана. При использовании модели порядка -1 вероятности символа присваивается определенное фиксированное значение, при этом все символы из кодируемого потока данных обычно считаются равновероятными.

Для точной оценки вероятности символа требуется учитывать контексты различных порядков. Оценки вероятности, полученные с использованием контекстов различных длин, объединяются в одну общую вероятность, после чего полученные результаты используются различными методами энтропийного сжатия в качестве весов или в качестве кодовых пространств кодируемых символов.

Алгоритм работы PPM может быть описан следующим образом. Для каждого контекста заводятся счетчики символов: если символ появляется в контексте, то его счетчик в данном контексте инкрементируется. К алфавиту кодируемой последовательности добавляется код ухода. Он также имеет свой счетчик, который в некоторых реализациях инкрементируется каждый раз, когда оценивается вероятность символа, еще не встречавшегося во входном потоке, а в других приравнивается некоторой фиксированной величине, например, единице. Вероятность ухода – это суммарная вероятность всех символов алфавита входного потока, еще ни разу не появлявшихся в

контексте. Она всегда отлична от нуля, кроме тех случаев, когда вероятность любого символа алфавита может быть оценена в рамках текущего контекста.

В первую очередь для некоторого символа рассматривается контекстная модель установленного максимального порядка M. Если данная модель оценивает вероятность рассматриваемого символа отлично от 0, то эта вероятность и используется каким-либо энтропийным методом для его кодирования. В противном случае выдается код ухода, и происходит рекурсивный переход к модели меньшего порядка. Уход к меньшим контекстам происходит до тех пор, пока либо вероятность символа не будет оценена, либо длина контекста не станет равняться -1, при которой гарантированно происходит оценка.

Наиболее эффективно и быстро PPM работает на задаче сжатия текстовых файлов. Среди его недостатков можно выделить медленное декодирование, низкая скорость обработки малоизбыточных данных, сложность выбора оптимального максимального порядка модели.

Данный алгоритм имеет множество модификаций и в том или ином виде используется во многих архиваторах, поскольку PPM лежит в основе в формата Rar.

1.1.7 LZ77

LZ77 – это первый из алгоритмов большого семейства LZ, представители которого обязаны своим появлением работам двух израильских исследователей – Якоба Зива и Абрахама Лемпела (Jacob Ziv & Abraham Lempel). Основная идея алгоритмов LZ заключается в использовании словаря просмотренных символов: если очередной считанный символ или группа символов содержится в словаре, то данное вхождение в сжатом файле заменяется указателем на запись словаря.

Кодирование сообщения в LZ77 происходит по мере движения «скользящего окна» по сообщению. Окно разделено на большую просмотренную часть сообщения (словарь) и значительно уступающую первой по размеру не просмотренную часть (буфер). Обычно размер окна составляет несколько килобайтов, из которых буфер занимает не более 100 байт. Алгоритм пытается найти наиболее длинное (ограниченное некоторым максимальным значением) вхождение подстроки из буфера в словарь. Если вхождение найдено, оно кодируется тройкой <смещение в словаре относительно начала подстроки, совпадающей с содержимым буфера; длина подстроки; первый символ в буфере, следующий за подстрокой>, после чего окно сдвигается на длину закодированной подстроки + 1 символ. Если же вхождение подстроки в словарь не было обнаружено, то алгоритмом возвращается код <0, 0, первый символ в буфере>.

Данный алгоритм очень прост в реализации, однако обладает целым рядом недостатков. Быстродействие алгоритма в значительной степени зависит от способа поиска подстроки в словаре: если поиск осуществляется полным перебором, то LZ77 будет работать предельно медленно. Еще с одной проблемой быстродействия алгоритма можно столкнуться при попытке увеличения степени сжатия – для этого необходимо увеличить размер окна, что приведет к пропорциональному уменьшению скорости работы метода. Также алгоритм LZ77плохо работает на разнородных данных – для кодирования однобайтового символа, не найденного в словаре, используется тройка <0, 0, символ>, которая занимает 3 байта вместо 1го, что в значительной степени понижает производительность алгоритма.

1.1.8 LZSS

Как уже было сказано выше, LZ77 породил целое семейство алгоритмов на своей основе. Одна из наиболее используемых в современных архиваторах модификаций оригинального алгоритма Лемпела-Зива – LZSS, разработанная Джеймсом Строером и Томасом Сжимански (James Storer & Thomas Szymanski) в 1982 году. Основными отличиями данного метода от оригинального LZ77 являются:

1. Хранение содержимого окна размера кратного степени 2 в циклической очереди;
2. Хранения словаря в виде двоичного лексикографически упорядоченного дерева поиска;
3. Кодирование метками, состоящими из 2х полей: <смещение; длина>;
4. Использование однобитного префикса для отличия незакодированных символов от пар <смещение; длина>.

Принцип работы кодера LZSS не сильно отличается от оригинального: на начальном этапе буфер наполняется символами и копированием содержимого буфера инициализируется дерево поиска. Основная итерация алгоритма состоит из 4 шагов:

1. Кодирование содержимого буфера;
2. Считывание информации в буфер, удаление устаревших строк из словаря;
3. Обновление дерева новыми строками, соответствующими считанным в буфер символам.

На третьем этапе вместе с добавлением новой строки в дерево происходит работа по поиску расположения и установлению длины максимального вхождения подстроки из

буфера в словарь. При добавлении строки в дерево, осуществляется последовательное перемещение от его корня к листу, выбор очередного поддерева происходит посредством лексикографического сравнения добавляемой строки и текущего узла, при этом запоминается длина совпадения строк и положение в дереве. Если на очередном шаге мы не обнаруживаем потомка, то он создается и заполняется соответствующей ссылкой на буфер. Если же содержимое буфера и строка, на которую ссылается текущий узел, совпадают, то ссылка в текущем узле обновляется. По завершению добавления строки кодеру известны как длина максимального вхождения, так и его смещение. Благодаря этому кодирование символов на следующей итерации легко осуществимо.

После того, как все символы сообщения были обработаны, в сжатый файл вставляется символ конца файла, который служит для декодера сигналом к прекращению работы.

Декодирование информации, сжатой по алгоритму LZSS, довольно тривиально. Анализируя первый бит сжатой информации, декодер определяет, следует ли за этим битом незакодированный символ или же пара <смещение, длина>. Если префикс сигнализирует о том, что далее в файле идет символ, то выдаются и помещаются в скользящее окно следующие 8 битов, в противном случае – соответствующее количество символов словаря.

1.1.9 LZMA

LZMA (Lempel–Ziv–Markov chain algorithm) – алгоритм сжатия информации, созданный Игорем Павловым в 2001г. Данный метод на первом этапе работы использует модификацию LZ77, обладающую возможностью использовать словари больших размеров, затем применяет арифметическое кодирование, используя сложную статистическую модель для предсказания каждого бита.

От классического LZ данный алгоритм отличается в первую очередь возможностью выбора окон большого размера – вплоть до 4 Гб, что позволяет более эффективно использовать доступные современным компьютерам объемы оперативной памяти. Последовательный перебор словаря при таких размерах сделал бы алгоритм непригодным к использованию, поэтому для хранения словаря используются такие структуры, позволяющие производить быстрый поиск, как цепочки хэш-таблиц и двоичные деревья.

1.1.10 LZH, LZARI

LZH (Lempel-Ziv Huffman) – алгоритм, разработанный японским исследователем Харуясу Йошизаки (Haruyasu Yoshizaki) в 1987 году. Данный метод является модификацией LZSS, которая использует коды Хаффмана для сжатия указателей, что приводит к незначительно лучшей степени сжатия, по сравнению с оригинальным алгоритмом.

LZARI (Lempel-Ziv-Arithmetic) – так же модификация LZSS, разработанная японским ученым Харухико Окумуро (Haruhiko Okumura). Данный метод отличается от LZH лишь тем, что для кодирования указателей используется не алгоритм Хаффмана, а арифметическое кодирование.

Данные алгоритмы используются в популярном в Японии архиваторе LHA.

1.1.11 Deflate

Формат Deflate был разработан Филом Кацем (Phil Katz) в 1993. Он специфицирует только работу декодера, сжатие осуществляет алгоритм типа LZH или любой другой алгоритм семейства LZ со скользящим окном, использующий определенные форматом Deflate коды Хаффмана для последующего кодирования литер и указателей.

Форматом Deflate специфицировано 3 типа сжатых данных:

1. Данные не сжатые вовсе – данный вариант используется, если информация, например, уже была компрессирована;
2. Данные, сжатые сначала алгоритмом LZ, затем алгоритмом Хаффмана. Двоичные деревья, используемые для компрессии, определяются форматом Deflate, таким образом, не требуется дополнительного пространства для их хранения. Этот метод называется методом статического кодирования Хаффмана, а коды Хаффмана, порождаемые им, называются фиксированными;
3. Данные, сжатые сначала алгоритмом LZ, затем кодированием Хаффмана с деревьями, которые создаются компрессором и хранятся вместе данными. Используемые метод и порождаемые коды Хаффмана называют динамическими.

Deflate-поток состоит из блоков, каждый из которых использует один из типов сжатия, описанных выше. Первые три бита блока – заголовок, определяющий метод кодирования данных и является ли блок последним. Содержательная часть блока в случае использования первого типа сжатия состоит из несжатой строки кодируемой

последовательности. В случае использования кодов Хаффмана блок содержит некоторую мета-информацию, описание двух таблиц Хаффмана, использованных для кодирования литералов и указателей на словарь вида <длина; смещение>, порожденных алгоритмом LZ, и непосредственно сами сжатые данные. Если данные кодируются динамическим методом Хаффмана, то деревья, порождаемые им, сжимаются фиксированными кодами Хаффмана, таблица которых задается форматом Deflate, и передаются в начале соответствующего блока данных.

В методе Deflate используется каноничный алгоритм Хаффмана, поэтому для получения кода символа достаточно знать длину этого кода. Таким образом, динамические коды Хаффмана могут быть описаны упорядоченной по значению кодируемых строк цепочкой длин кодов. Такие цепочки кодируются алгоритмом Хаффмана еще раз и также описываются упорядоченной последовательностью длин кодов, которые задаются с помощью 3-битовых чисел.

Окно алгоритма LZ может распространяться на несколько блоков, но имеет ограничение на величину смещения в 32 Кб. Так же вводится ограничение на максимальный размер совпадающей строки в 258 байт. Выбор этого значения коррелирует с тем, что литералы и длины совпадений объединены в единый алфавит со значениями {0 .. 285}, где подмножество {0 .. 255} отводится под литералы, значение 256 выступает в качестве символа конца блока, а набор {257 .. 285} определяет длины совпадений. Причем коды длины могут быть комбинацией числа из приведенного выше диапазона и от 0 до 5 дополнительно считываемых битов.

Отдельно от литералов и длин кодируются смещения. Схема их кодирования походит на схему кодирования длин: используется значение базы {0 .. 29} и дополнительно считывается от 0 до 13 бит, таким образом покрываются значения от 1 до 32768. Объединения смещений в группы обуславливается повышенной эффективностью работы декодера алгоритма Хаффмана на порождаемых таким образом кодах.

1.2 Тестирование решений на основе архивации

В случае использования в системе хранения данных в качестве средств сжатия решения на основе архивации, то имеет смысл рассматривать 2 способа хранения образов ВМ в архивах. Первый способ заключается в том, что для каждой виртуальной машины создается отдельный архив. Второй подразумевает хранение виртуальных машин, сгруппированных по степени совпадения, в непрерывных архивах. Второй способ дает значительный прирост степени сжатия при использовании словарных методов с большим размером словаря, однако он обладает критичным недостатком: извлечение отдельного

файла из середины или конца архива происходит медленнее, чем из его начала, поскольку для этого приходится анализировать все предыдущие упакованные файлы. Так как степень сжатия ВМ, при использовании первого способа, остается приемлемой для системы, было принято тестировать архиваторы именно в этом режиме работы.

Для тестирования с помощью менеджера виртуальных машин VirtualBox были созданы 2 тестовых образа виртуальных машин формата vmdk, с объемом виртуального жесткого диска и, как следствие, размером файла-образа равными 5 Гб. Первый образ поставлялся с «чистой» установкой Windows XP SP3 (windows-origin.vmdk), второй – с той же копией Windows XP SP3 и полной установкой пакета офисных приложений MS Office (windows-office.vmdk). Отказ от динамического виртуального жесткого диска был выполнен в силу того, что облачная инфраструктура XCP, лежащая в основе MCD, для импорта виртуальных машин извне требует образы в формате raw, приведение к которому возможно только в случае использования «статического» виртуального диска. Для конвертирования образов из одного формата в другой можно воспользоваться утилитой `qemu-img` с опцией `convert`.

Тестовым заданием являлись последовательные упаковка и распаковка образов ВМ с различными конфигурациями архиваторов. Тестирование на компьютере со следующими техническими характеристиками:

Процессор: Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz;

RAM: 4GiB, 1333MHz;

HDD: 500Gb, 7200rpm, ср. скорость чтения: 111.70 MB/s. ср. скорость записи: 114.22 MB/s.

Поскольку система MCD не накладывает никаких ограничений на используемую в хранилище ВМ операционную систему, были протестированы архиваторы как для ОС семейства Linus, так и для ОС семейства Windows. Однако на предварительном этапе тестирования было принято решение отказаться от дальнейших работ с программными продуктами для ОС Windows, поскольку на задаче декомпрессии они показывали скорость в 2-3 раза хуже, чем их полные аналоги для ОС Linux, в силу скоростных возможностей используемых файловых систем – NTFS и EXT3 соответственно. В пользу этого решения так же повлияло то, что XCP объединяет в облачную инфраструктуру хосты под управлением ОС семейства Linux и позволяет использовать их не только в качестве машин для запуска ВМ, но и под хранение их образов.

Дальнейшее тестирование архиваторов производилось на компьютере под управлением Ubuntu 13.10 x64. В процессе работы архиваторов для операций компрессии и распаковки (как на жесткий диск, так и на устройство `/dev/null`, что позволяет

оценить производительность архиваторов без влияния затрат на работу с диском по записи) производились замеры времени выполнения (функции `time`, `date`), степени загрузки процессора (с помощью утилиты `atop`), оперативной памяти, жесткого диска (утилита `iotop`), а также степени сжатия файла (`stat c%s`) при различных вариантах конфигурации средств сжатия.

Команда `time` запускает указанную команду с заданными аргументами. По завершении работы команды, `time` выводит статистическое сообщение об использованном командой времени при этом запуске: фактически затраченном времени; времени CPU, которое процесс провел в режиме ядра; времени CPU, которое процесс провел в режиме пользователя.

Утилита `atop` является интерактивным монитором производительности, который позволяет отслеживать нагрузку процессора, оперативной памяти, жесткого диска, а также распределение нагрузок по обрабатываемым процессам. Запустив профилирование системы с периодом в 1 секунду, применив необходимые фильтры, можно получить подробную статистику использования процессом сжатия физических ресурсов ПК.

Инструмент для получения подробной информации по использованию жесткого диска `iotop` показывает пользователю степень загруженности каналов ввода-вывода. Данная утилита была использована для определения влияния загруженности каналов ввода-вывода на скорость работы компрессоров и декомпрессоров.

Команда `stat c%s` была использована для получения информации о размере сжатого файла, полученного в результате работы компрессора.

Были протестированы следующие архиваторы:

Windows (отсеяны на предварительном этапе тестирования):

- WinRAR (PPMII + LZSS)
- WinZip (LZ77 + Huffman)
- 7-zip (LZMA)

Linux:

- rar (PPMII + LZSS)
- zip (LZ77 + Huffman)
- 7-zip (LZMA)
- bzip2 (BWT + MTF + Huffman)
- pbzip2 (BWT + MTF + Huffman)
- gzip (LZ77 + Huffman)

- pigz (LZ77 + Huffman)
- lzop (LZO)

Также были проведены тесты утилита cp, которая являлась своеобразным скоростным эталоном со степенью компрессии равной 1.

По результатам тестирования было определено, что наиболее высокую скорость декомпрессии с допустимым значением степени сжатия информации на задаче архивации образов виртуальных машин показывает архиватор RAR. Время распаковки файла-образа ВМ с MS Office составляет в среднем 68 секунд на всех степенях сжатия, что является довольно неплохим результатом в сравнении с эталонным временем копирования равным примерно 54 секунд. Степень сжатия также является удовлетворительной и составляет 4,1. Однако данный архиватор является условно-бесплатным.

Среди бесплатных архиваторов наилучшие результаты показали lzop и pigz: время декомпрессии заархивированного файла ВМ составляет в среднем 81 и 70 секунд соответственно. Из недостатков данных архиваторов можно выделить более низкую степень сжатия у lzop равную ~3,5 (в сравнении с 5,3 у 7z –наилучшего архиватора по степени сжатия) и высокую нагрузку на процессор у pigz на операции архивации (~300% из максимума в 400%, обусловленного использованием 4-ядерного процессора, в сравнении с 47% у lzop – самого экономичного с точки зрения потребления ресурсов ЦП). Подробные результаты тестирования могут быть найдены в приложении.

1.3 Решения на основе дедупликации

Дедупликация – это специальный метод сжатия информации, который для уменьшения объема занимаемого данными пространства использует удаление дублирующих копий повторяющихся единиц информации.

Можно выделить два основных вида дедупликации – дедупликация на уровне файлов и блочная дедупликация. В случае дедупликации на уровне файлов единицей данных является отдельный файл, при этом сжатие информации производится за счет исключения дублирующих файлов из системы хранения данных. Примером файловой дедупликации может считаться механизм символических ссылок.

В методе блочной дедупликации единицей дедупликации является блок данных произвольной длины, называемый чанком (chunk). В процессе дедупликации производится проверка блоков данных на уникальность и, в случае успешного прохождения проверки, их сохранение. Если же блок данных совпадает с уже содержащимся в памяти, то он заменяется ссылкой на ранее запомненный элемент;

дубликат при этом удаляется. Зачастую для проверки уникальности используются хэш-функции – в системе дедупликации хранится хэш-таблица всех уникальных блоков данных. Если обнаруживается совпадение хэшей для разных блоков данных, то они считаются идентичными и сохраняются в единственном экземпляре с набором ссылок на этот экземпляр. Блоки данных могут быть постоянной длины фиксированного размера или переменной, когда размер чанка подбирается в зависимости от дедуплицируемых данных динамически. Стоит заметить, что наибольшая эффективность сжатия за счет дедупликации достигается при уменьшении размера блока, а значит максимизации коэффициента его повторяемости.

Ввиду специфики системы MCD, далее будет рассматриваться лишь блочная дедупликация, так как при хранении виртуальных машин, каждая из которых отличается уникальной предустановленной программой, искать идентичные файлы образов не имеет смысла, зато процент совпадающих блоков может стремиться к 100.

Существуют два ключевых метода обработки данных при дедупликации. Первый из них – inline-дедупликация (поточный метод). При использовании данного метода дедупликация данных происходит «налету» – непосредственно перед записью на диск. Если же используется post-process (пост-процессный) метод, то анализ и обработку данных производится только после их копирования на устройство хранения. Каждый из методов обладает своими достоинствами и недостатками: при inline-дедупликации в значительной степени снижаются требования к доступным объемам долговременной памяти, поскольку недедуплицированные данные не записываются на диск, однако при этом возрастает нагрузка на процессор и оперативную память. В свою очередь при использовании пост-процессного метода данные сначала записываются на диск, после чего запускается процесс их дедупликации, а это значит, что, в зависимости от политики хранения данных, они могут быть сразу доступны для манипуляций. Однако сама дедудпликация будет требовать изначально больших дисковых ресурсов и временных затрат для всего процесса

1.3.1 SDFS

SDFS – это дедуплицирующая распределенная файловая система, разработанная в рамках проекта Opendedup, изначальной целью которого являлось создание файловой системы для хранения образов виртуальных машин. Данная ФС реализована с использованием механизма создания файловых систем пространства пользователя FUSE. В состав SDFS входят непосредственно файловая система уровня пользователя, с помощью которой осуществляется файловый доступ к данным; сервис обрабатывающий

запросы от файловой системы на запись и чтение файлов, а также отвечающий за хранение карты дубликатов и различных метаданных; движок дедупликации, который занимается непосредственно манипуляциями с данными и хранением хэш-таблицы. Выявление и устранение дубликатов данных может происходить на уровне блоков как фиксированного размера (4К,8К,16К,32К,64К,128К), так и переменного.

Для использования данной ФС необходимо наличие дистрибутива Linux x64, предустановленного модуля Fuse, пакета Java, а также минимум 2 Гб оперативной памяти. Стоит заметить, что для обеспечения конкурентоспособной скорости доступа хэш-таблицы хранятся в оперативной памяти, в результате чего на каждый терабайт дедуплицируемых данных требуется дополнительно ~500 Мб RAM.

1.3.2 LessFS

LessFS – файловая система пространства пользователя (FUSE-based). Основная идея данного решения заключается в использовании механизмов дедупликации и компрессии для уменьшения объемов использования дискового пространства. Для более гибкой работы системы доступны несколько фиксированных размеров блока, также разработчиками предоставлен довольно большой выбор методов компрессии – до проведения дедупликации данные могут быть сжаты алгоритмами QuickLZ, Snappy, Bzip2, Gzip или Deflate. Одной из возможностей данной ФС является шифрование данных.

Как и в случае с SDFS, часть необходимой для работы системы информации хранится в оперативной памяти, что накладывает требование на наличие 500 Мб оперативной памяти на каждый терабайт дедуплицируемых данных. Требования к программному обеспечению также схожи: Linux x64, Fuse.

1.3.3 ZFS on Linux

ZFS on Linux – это порт известной файловой системы ZFS (Zettabyte File System), созданной компанией Sun Microsystems для ОС Solaris, в качестве модуля ядра Linux. ZFS обладает возможностями хранения больших объемов данных (позволяет адресовать до 3E26 ТБ данных), управления снапшотами и пулами хранения, обеспечения контроля целостности данных и дедупликации, журналирования, предоставления квот. Также ZFS поддерживает избирательное сжатие данных за счет использования библиотек LZJB, Gzip, ZLE. Таким образом, Zettabyte File System комбинирует в себе одновременно функциональность файловой системы, менеджера томов, программного RAID-контроллера, средства дедупликации, компрессии и модуля резервного копирования данных.

Для возможности и работы с ZoL требуется наличие 64-битного дистрибутива Linux и минимум 2 Гб оперативной памяти. На каждый дедуплицируемый терабайт данных система дополнительно требует до 5 Гб RAM.

1.4 Тестирование решений на основе дедупликации

В силу того, что среди свободных решений были найдены лишь файловые системы, требующие установленной системы семейства Linux, тестирование проводилось на компьютере под операционной системой Ubuntu 13.10. Технические характеристики ПК были описаны в разделе, посвященном архивации.

Для тестов помимо двух описанных образов виртуальных машин (windows-origin и windows-office) были созданы еще 2 файла ВМ, каждый размером в 5 Гб, первый из которых был получен из образа windows-office незначительными изменениями конфигурации MS Office (windows-office-changed), а второй был получен из windows-origin путем установки OpenOffice (windows-open-office). Файлы-образы были конвертированы в формат raw, после чего производилось последовательное копирование образов с раздела с файловой системой EXT3 на раздел с ФС с дедупликацией в порядке windows-origin, windows-office, windows-office-changed, windows-open-office. Также производилось дублирование образа на разделе тестируемой ФС и восстановление этого образа, путем копирования его из раздела с тестируемой ФС на раздел жесткого диска с EXT3.

На каждом шаге работы теста производились измерения времени копирования (`time` к операции копирования, `date` непосредственно перед копированием и после него), загрузки процессора, жесткого диска и оперативной памяти (вызов `atop` с периодом в 1 секунду, вызов iotop с периодом равным 0,5 секунды), а также степени дедупликации и компрессии. Для измерения последних двух характеристик в конце каждой операции копирования вызывались специфичные для файловых систем средства анализа. Для ZoL это команды

- `zfs list [%dataset%]` – для просмотра базовой информации о наборе данных; по этой команде отображаются имена всех наборов данных в системе с подробной информацией о них или, в случае указания набора, подробная информация о наборе, включающая значения свойств `used` (объем пространства, занимаемого набором данных и всеми дочерними элементами) и `available` (пространство, доступное для набора данных и всех его дочерних элементов при условии отсутствия других операций с пулом);

- `zfs get compressratio %dataset%` – определяет достигнутую степень сжатия для набора данных; рассчитывается на основе логического размера всех файлов и объема связанных физических данных;
- `zpool get dedupratio %pool%` – определяет достигнутую на пуле степень дедупликации данных.

Чтобы посмотреть подробную статистику по дедупликации в SDFS необходимо воспользоваться командой `sdfscli.sh --volume-info`, которая вернет округленную до Гб оценку объема уникальных и дуплицированных данных, реально используемого дискового пространства и прочее. Для получения подобной информации в LessFS не предусмотрено специальной утилиты, однако требуемую информацию все равно можно получить посредством использования команды `df -t fuse.lessfs`, которая предоставит отчёт об использовании дискового пространства файловыми системами типа `fuse.lessfs`.

Для рассмотренных файловых систем были протестированы все возможные комбинации размеров блока и способов предварительного сжатия информации.

Как и прежде, наиболее интересными характеристиками являлись время восстановления файла (скорость чтения данных с раздела с тестируемой ФС) и объем файла после проведения дедупликации. Самым эффективным решением по результатам оказался порт ZFS, который показывал как отличное время восстановления (1 минута против 2-10 минут SDFS/LessFS), так и отличную степень сжатия лишь при использовании дедупликации, без использования компрессии. Такая разница в скорости восстановления данных наблюдается в силу того, что коды SDFS/LessFS исполняются в в режиме пользователя, тогда как кода ZoL исполняется в ядре. Подробные результаты тестирования могут быть найдены в приложении.

Однако за все приходится платить. Для ZFS на каждый Тб дедуплицируемых данных необходимо иметь 5 Гб оперативной памяти. Требуемый для SDFS и Lessfs объем оперативной памяти рассчитывается по формуле (volume size/chunk size)*33 и составляет всего порядка 500 Мб на 1 Тб дедуплицируемой информации. Объем требуемой оперативной памяти может быть уменьшен за счет использования в качестве хранилищ хэш-таблиц и карт дедупликации высокоскоростных SSD, что, однако, снова требует дополнительных материальных вложений на покупку и поддержку оборудования и несколько влияет на производительность системы хранения.

Глава 2. Разработка спецификации файловой системы специального назначения

Как было показано выше, решения архивации и дедупликации достаточно хорошо решают задачу сжатия информации, однако производительность операции восстановления файлов сильно зависит как от выбранного алгоритма, так и в значительной степени от возможностей процессора и, конечно же, устройства хранения. Среди недостатков файловых систем с дедупликацией также можно выделить высокие требования к объему доступной оперативной памяти – соотношение до 5 Гб RAM на 1 Тб дедуплицируемых данных. Исходя из перечисленных недостатков, было принято решение разработать спецификацию своей файловой системы, специализированную с одной стороны на компактное хранение образов виртуальных машин, а с другой на скоростное порождение копий и высокую скорость чтения данных, малотребовательную при этом к системным ресурсам.

Основная идея скоростного копирования данных была позаимствованная у системного вызова `fork` операционной системы Linux, который создает новый процесс-потомок, являющийся практически полной копией выполняющего этот вызов процесса-родителя. Особенность системного вызова `fork` в ОС Linux заключается в том, что вместо копирования страницы памяти родительского процесса происходит их отображение в новый процесс, новые же страницы памяти создаются только при изменении её содержимого одним из процессов. Данная техника называется copy-on-write – копирование при записи.

Ввиду сложности разработки файловой системы «с нуля», было принято решение взять за основу одну из ФС с открытой спецификацией. В качестве такой системы была выбрана FAT, в виду своей простоты, популярности, и применимости концепции цепочек кластеров к задаче реализации файлового аналога `fork`.

Таким образом, была поставлена задача создания спецификации файловой системы на основе FAT, обладающей файловым аналогом операции `fork`. Суть данной операции заключается в моментальном создании точной копия файла на основе ссылки на оригинальный файл. Все изменения над образованным операцией `fork` файлом, как и над файлом, на который ссылается хоть один потомок, не должны затрагивать оригинальный файл.

Как известно, в FAT файлы могут быть условно разделены на 2 группы: директории и собственно файлы. Директория FAT – ни что иное как файл, состоящий из

списка 32-байтных структур двух типов: `long filename` (длинные имена файлов) и `alias` (псевдонимы, алиасы).

Смещение	*Длина в байтах*	*Описание*
00	8	Имя файла
08	3	Расширение
0B	1	Атрибуты
0C	**1**	**Зарезервировано**
0D	1	Время создания в мс
0E	2	Время создания
10	2	Дата создания
12	2	Дата последнего обращения
14	2	Старшее слово номера первого кластера
16	2	Время последней записи
18	2	Дата последней записи
1A	2	Младшее слово номера первого кластера
1C	4	Размер файла

Табл 2.1. Структура алиаса

Для реализации copy-on-write на основе FAT предлагается любой файл в системе рассматривать как структуру, состоящую из алиаса на некоторую исходную цепочку и набора групп алиасов на произведенные с исходной цепочкой изменения. Можно провести аналогию с папкой, содержащей исходный файл и набор «заплаток» для этого файла, сгруппированных по времени их создания. В качестве предложенной выше структуры можно использовать обертку типа «`wrap`» на основе файлов типа «`directory`». Для определения типа «`wrap`» можно воспользоваться одним из зарезервированных битов в списке атрибутов.

Значение	*Описание*
01	Только для чтения
02	Скрытый
04	Системный
08	Корневая директория
10	Директория
20	Сжатый
40	**Зарезервировано**
80	**Зарезервировано**

Табл 2.2. Структура поля Atribute

В качестве первого алиаса в псевдо-папке предлагается использовать алиас на оригинальную цепочку, либо на «`wrap`»-родитель, в случае образования файла в результате операции `fork`. Данный алиас можно отделить от алиасов изменений за счет использования зарезервированного байта.

Также, если «wrap» образован в результате выполнения операции fork, то у алиаса, ссылающегося на «wrap»-родитель, необходимо выставить атрибут «link», для отделения «ссылок» от оригинальных файлов. Флаг данного атрибута может быть выполнен на основе зарезервированных атрибутных полей.

В качестве имени файла предлагается рассматривать имя псевдо-директории, содержащей файл.

Файлы в системе могут быть в двух состояниях: когда они не являются родителем ни для одного другого файла, либо когда на них ссылается хоть один файл-потомок. В то же время можно выделить другие две группы файлов: файлы, образованные в результате операции fork и все остальные.

Если на текущее состояние файла нет ни одной ссылки, то модификации могут производиться над последней группой цепочек изменений. Если файл является «простым» (то есть он был создан не в результате fork-а), на него нет ссылок и у него нет ни одной цепочки с изменениями, то может производиться модификация оригинальной цепочкой.

Если же на текущее состояние есть хоть одна ссылка или же файл образован в результате операции fork, то изменения в файле должны порождать отдельную группу цепочек кластеров с требуемыми модификациями и соответствующими алиасами в «wrap»-файле.

Для отслеживания количества ссылок на оригинальную последовательность предлагается использовать механизм счетчиков, реализовать который можно на базе структур типа long filename, из идущих перед алиасом слотов которой составляются длинные имена файлов в FAT32:

Смещение	*Длина в байтах*	*Описание*
00	1	Номер слота в последовательности
01	**10d**	**1-5 символы длинного имени**
0B	1	Атрибуты (содержит атрибут длинного имени)
0C	1	Зарезервировано
0D	1	Контрольная сумма алиаса
0E	**12d**	**6-11 символы длинного имени**
1A	2	Зарезервировано
1C	**4**	**12-13 символы длинного имени**

Табл 2.3. Структура слота Long filename

Предлагается для счетчиков использовать поля символов длинного имени подобных структур. Для отделения структур счетчиков от структур длинного имени можно воспользоваться значением зарезервированных полей. Таким образом, процесс изменения значения счетчика будет сродни процессу переименованию файла.

2.1 Fork измененного файла

Отдельно стоит рассмотреть операцию `fork`, где в качестве файла-родителя выступает измененный файл, изменения в котором выполнены в виде групп алиасов на цепочки модификаций. Данный случай интересен тем, что после добавления ссылки на такой файл-родитель, последующие его изменения породят новые группы алиасов на очередные цепочки изменений, которые должны игнорироваться при чтении файла, порожденного в результате `fork`. Для отделения нужных изменений от ненужных, предлагается для файла-потомка хранить количество групп изменений, которые требуется накатить поверх оригинальной цепочки. Для хранения количества изменений, по аналогии с хранением значений счетчиков ссылок, можно использовать модифицированные структуры `long filename`, их же предлагается использовать в качестве разделителя групп изменений.

2.2 Основные операции

2.2.1 Добавление файла

Добавление файла можно рассматривать как последовательное выполнение несколько модифицированных стандартных операций системы FAT32 – создание папки и добавление записи в папку. Модификация алгоритма создания папки заключается в отказе от создания записей «`.`» и «`..`», а также в установке атрибута «`wrap`». Модификация алгоритма добавления записи заключается в выделении памяти не только под длинное имя и алиас, но и под счетчик ссылок на файл.

2.2.2 Удаление файла

Удаление файла считается невозможным, если на него есть хоть одна ссылка. Алгоритм удаления файла состоит из 2х основных этапов: удаление записей внутри псевдо-директории и удаление самой псевдо-директории. В случае если удаляемый файл является потомком, счетчик файла-источника должен быть уменьшен на единицу.

2.2.3 Fork

Алгоритм `fork`-а заключается в создании псевдо-директории типа «`wrap`» с требуемым значением количества применяемых изменений, которые записываются перед длинным именем файла в модифицированных слотах `long filename`; добавлении алиаса на файл-источник с указанием атрибута «`link`» и выделением места под счетчик ссылок на данный файл; увеличении счетчика ссылок у файла-источника

2.2.4 *Изменение файла*

Если в структуре «wrap» файл обладает атрибутом «link» или счетчик ссылок на него > 0, то он считается заблокированным и изменение производится в виде создания новых цепочек кластеров, и размещения алиасов на них в «wrap»-файле, либо в виде модификаций цепочек изменений текущего состояния файла. Изменения группируются по мере появления ссылок на текущее состояние файла. В случае, когда файл не заблокирован, изменения могут производиться непосредственно над исходной цепочкой.

2.2.5 *Чтение файла*

Чтение производится путем считывания исходной цепочки с последовательным применением групп цепочек-изменений.

Спецификация ФС может быть найдена в приложении.

Глава 3. Статическая модификация образов виртуальных машин для внедрения автозапуска приложений

Процесс запуска приложений в системе MCD состоит из этапов создания пользовательской копии виртуальной машины с запускаемым приложением, внедрения автозапуска этого приложения с опциональным открытием некоторого файла, а также внедрения автозапуска клиента X11 со всеми необходимым для взаимодействия с конкретным пользователем настройками.

Таким образом, задача запуска приложения в системе MCD может быть разделена на 3 подзадачи:

1. Исследование возможностей автозапуска приложений в различных ОС;
2. Изучение техник внедрения автозапуска в виртуальную машину;
3. Поиск метода доставки обрабатываемых пользователем файлов в ВМ.

3.1 Автозапуск приложений

3.1.1 Автозагрузка приложений в ОС семейства Windows

Задача автозапуска приложений в ОС семейства Windows может быть решена двумя основными способами: модификацией реестра или размещением в директории автозапуска ярлыка на требуемое приложение или bat-файла, с требуемыми командами.

Существует множество возможностей внедрения автозагрузки приложения путем модификации реестра, которые подробно описаны в литературе по теме безопасности информационных систем. Наиболее простой из таких возможностей является внесение полного пути к исполняемому файлу в ветвь системного реестра:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run.`

Размещение ярлыка приложения или bat-файла для их автозапуска на старте системы осуществляется по умолчанию в папку
`[C]:\Users\All Users\Microsoft\Windows\Start Menu\Programs\Startup`
, где `[C]` – буква диска, на который установлена операционная система. В случае необходимости путь к папке автозагрузки может быть изменен модификацией записи «`Common Startup`» ветви системного реестра
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders`

Использование механизма bat-файлов удобно как для запуска приложений, так и для открытия файлов. Для этих задач достаточно воспользоваться командой `start`, простейший вид которой: `start путь-к-программе [путь-к-открываемому-файлу]`

3.1.2 *Автозагрузка приложений в ОС семейства Linux*

В операционной системе Linux реализация автозапуска приложений также может быть решена несколькими способами.

Первый из способов заключается в добавлении команды запуска того или иного приложения в `rc.local`. Примером команды запуска приложения может являться `cd /path/to/soft ; ./script`. В случае использования `rc.local`, необходимо убедиться, что конфигурационный файл `rc.conf` содержит строку `local_enable="YES"`.

Второй способ – размещение шел-скрипта с инструкциями для запуска программы в каталог `/etc/init.d` и добавление символической ссылки на этот шел-скрипт в директорию в `/etc/rc[runlevel].d`, где `[runlevel]` – целое число от 0 до 6 (в некоторых дистрибутивах – от 0 до 9), обозначающее уровень выполнения программы. Символическая ссылка должна иметь имя вида `S№№имя_скрипта`, где №№ – это номер очередности запуска программы. Образец шелл-скрипта может быть найден в `/etc/init.d/skeleton`.

Способ под номером три заключается в использовании файла `inittab`. Данный файл состоит из строк вида `id:runlevels:action:process`, где

- `id` – идентификатор строки, состоящий из 1-4 символов. В `inittab` не может быть двух строк с одинаковыми идентификаторами;
- `runlevels` – уровни выполнения, задаваемые цифрами или буквами без разделителей, на которых будет запущен представленный данной строкой процесс;
- `process` – имя программы, вызываемой при переходе на указанные уровни выполнения;
- `action` – действие, которое определяет дополнительные условия выполнения процесса.

Допустимыми значениями поля action являются:

- `respawn` – процесс перезапускается в случае завершения его работы;
- `once` – процесс вызывается единожды при переходе на указанный уровень;
- `wait` – процесс запускается один раз при переходе на указанный уровень, при этом `init` ожидает его завершения для продолжения своей работы;
- `sysinit` – процесс запускается до перехода на какой-либо уровень выполнения, до запуска процессов, помеченных словами `boot` и `bootwait`;

- `boot` – запуск процесса осуществляется на этапе загрузки системы независимо от указанного уровня выполнения;
- `bootwait` – процесс запускается на этапе загрузки системы независимо от уровня выполнения, `init` ожидается завершения его работы;
- `initdefault` – строка определяет уровень выполнения по умолчанию, поле `process` при этом игнорируется;
- `off` – строка игнорируется;
- `powerwait` – `init` останавливает систему, когда пропало питание, и компьютер перешел на использование UPS;
- `ctrlaltdel` – разрешает перезагрузку системы по нажатию комбинации клавиш Ctrl + Alt + Del.

Таким образом, добавив в `inittab` необходимые строки с именами программ и требуемыми параметрами исполнения, можно довольно тонко настроить процесс автозапуска приложений в ОС Linux.

3.1.3 Автозагрузка приложений в Mac OS

Автозапуск в Mac OS может быть реализован на основе механизма запуска сервисов с помощью утилиты `launchd`. Сервисы представляются в системе в виде конфигурационных XML-файлов формата .plist, которые на запуске системе обрабатываются и подаются на исполнение указанной выше утилитой. Конфигурация сервиса, в зависимости от его назначения, может быть расположена в одной из нескольких директорий:

- `[Userdir]/Library/LaunchAgents`

Конфигурационные файлы, находящиеся в данной директории, выполняются при входе в систему пользователя, в домашнем каталоге которого расположена папка с конфигурациями. Исполнение происходит с правами вошедшего в систему пользователя;

- `/Library/LaunchAgents`

Находящиеся в данной системной директории конфигурационные файлы обрабатываются и исполняются при входе систему любого из пользователей с его правами.

- `/Library/LaunchDaemons`

Конфигурации сервисов, расположенных в данной системной директории, запускаются от лица администратора (root) до момента входа пользователя в систему.

- `/System/Library/LaunchAgents`

Конфигурационные файлы, находящиеся в данной директории, являются системными, запускаются при входе систему любого пользователя;

- `/System/Library/LaunchDaemons`

Конфигурационные файлы, находящиеся в данной директории, являются системными, запускаются с правами администратора до входа пользователя в систему.

Конфигурационные файлы не являются самостоятельными программами, они используются для моделирования сценариев загрузки программ. В простейшем случае, .plist-файл должен выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>ServiceName</string>
  <key>ProgramArguments</key>
  <array>
     <string>path-to-program</string>
     <string>arg1</string>
     <string>arg2</string>
     ...
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

В данном XML-файле основными ключами являются `Label` (параметр, идущий после этого ключа, является названием сервиса, отображаемым в системе); `ProgramArguments` (массив, следующий за этим ключом, содержит в первой строке путь к исполняемому файлу, а в последующих – параметры его запуска); `RunAtLoad` (значение, идущее после этого ключа, указывает, должен ли сервис исполняться при загрузке ОС).

После создания конфигурационного файла для автозапуска программы, его требуется разместить в одной из описанных выше директорий и установить необходимые права доступа.

3.2 Внедрение автозапуска приложений в ВМ

Проблема внедрения автозапуска приложений в виртуальную машину решается путем монтирования файла-образа ВМ, при котором происходит предоставление пользователю доступа к файлам и папкам виртуального диска из точки монтирования. Для монтирования файлов ВМ создано большое разнообразие специальных утилит, одной из наиболее простых и быстрых в использовании является `guestmount`, разработанная в рамках проекта libguestfs.

Для монтирования ВМ с помощью утилиты `guestmount` в простейшем случае необходимо воспользоваться командой следующего вида:

```
guestmount -a guest.img -i --rw /mnt
```

где `-a guest.img` – опция для образа ВМ, `-i` – опция для автоматического поиска разделов виртуального диска, `-rw` – опция для открытия файла для чтения-записи, `mnt` – точка монтирования.

После того, как разделы виртуального диска появились в системе как простые папки, достаточно изменить исходные (например, файл реестра) или же скопировать в требуемое место необходимые для автозапуска данные (например, bat-файл в папку автозагрузки системы). По завершении операций с виртуальным диском ВМ образ демонтируется командой `guestunmount`.

3.3 Работа с внешними файлами

Задачи безопасного взаимодействия изолированного в виртуальной машине приложения с внешними данными разных уровней секретности заслуживает особого внимания. Требуется обеспечить защищенное соединение к хранилищу пользовательских данных, которое предоставляет многопользовательский доступ данным, контролируемый принятой политикой безопасности.

В результате проведения предварительного исследования в данной области был найден WebDAV – протокол высокого уровня, работающий поверх HTTP. Среди его ключевых особенностей можно выделить встроенный механизм блокировок и версионирования для реализации конкурентного доступа к файлам; поддержку основных файловых операций; широкую распространенность и наличие встроенных средств работы с ним в операционных системах семейств Windows, Linux, Mac OS; высокую скорость

работы; возможность передачи метаданных вместе с непосредственно файлом; расширяемость.

Немаловажным аргументом в пользу выбора WebDAV в качестве сетевого протокола для взаимодействия с хранилищем документов является успешный опыт его использования в схожих задачах крупных ИТ компаний. Например, данный WebDAV был выбран программистами компании Яндекс для взаимодействия клиентских приложений с хранилищами Яндекс.Диска.

Глава 4. Разработка прототипа модуля эффективного хранения образов виртуальных машин с возможностью их модификации для автозапуска приложений

В этой главе рассмотрены основные аспекты построения прототипа модуля эффективного хранения образов виртуальных машин, обладающего функциональностью внедрения в них автозапуска приложений.

4.1 Базовая функциональность

Доказательство возможности существования модуля эффективного хранения образов ВМ сводится к созданию клиент-серверного приложения, работающего в облачной инфраструктуре XCP, в рамках работы которого:

1. По команде клиента на стороне сервера из некоторого хранилища осуществляется извлечение (с допустимой скоростью) образа ВМ с требуемым приложением;
2. Производится внедрение автозапуска данного приложения в эту ВМ;
3. При необходимости происходит процесс конвертации образа в необходимый для гипервизора Xen вид;
4. Модифицированная виртуальная машина разворачивается на сервере;
5. Клиенту отправляется сообщение об успешном запуске ВМ.

4.2 Технологии

В качестве основного языка программирования был выбран PHP в силу его простоты, гибкости, серверной ориентированности, удобства работы с сокетами, большого количества готовых библиотек для широкого спектра задач.

Для написания вспомогательных модулей сервера был выбран язык сценариев командной оболочки bash, на котором очень естественно выражаются задачи инициализации файловых систем, последовательности команд для разворачивания виртуальных машин.

4.3 Архитектура

Модуль эффективного хранения состоит из back-end-а, осуществляющего обработку сообщений от клиента, общающегося с ним посредством механизма сокетов, который определяется правила сетевого взаимодействия. Команда по запуску приложения

разбирается, осуществляется поиск требуемого образа виртуальной машине в хранилище, после чего образ извлекается взаимодействия с Xen. Модуль внедрения автозапуска осуществляет модификацию образа виртуальной машины в зависимости от используемой в ней операционной системы.

Для proof-of-concept в качестве метода хранения была выбрана файловая система SDFS, обладающая высокой скоростью восстановления файлов, приемлемой степенью сжатия и низкими требованиями к объему доступной оперативной памяти. Ввиду модульной организации, можно легко перейти на любой из описанных в работе методов хранения.

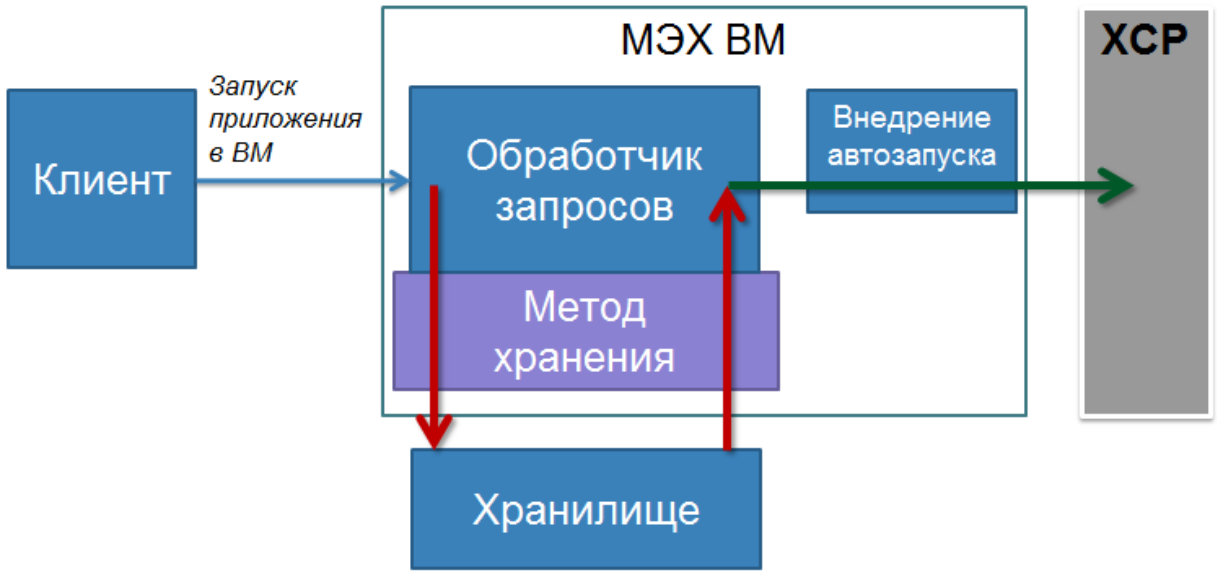


Рис. 4.1. Архитектура модуля эффективного хранения ВМ

Заключение

В рамках данной работы была реализована система эффективного хранения виртуальных машин с возможностью их модификации для автозапуска приложений. Ниже представлен детальный список результатов:

1. Изучены ключевые подходы сжатия;
2. Сделан подробный обзор основных алгоритмов компрессии без потери данных и дедупликации;
3. Проведены тестирование и сравнительный анализ существующих программных решений на основе технологий архивации и дедупликации;
4. Разработан пример спецификации файловой системы, специализирующейся на высокой скорости доступа к информации и эффективном использовании дискового пространства;
5. Разработан прототип модуля эффективного хранения образов виртуальных машин с функцией внедрения автозапуска.

Работа в рамках проекта MCD была представлена на следующих конференциях:

- СПИСОК-2014;
- Научно-техническая конференция студентов «СТУДЕНЧЕСКАЯ НАУЧНАЯ ВЕСНА»;
- 3-я международная научно-техническая конференция «Аэрокосмические технологии», посвященная 100-летию со дня рождения академика В.Н.Челомея.

Дальнейшее развитие

Приоритетным направлением дальнейшего развития темы данной дипломной работы является задача интеграции модуля эффективного хранения в систему MCD, для реализации которой можно выделить ряд подзадач. Во-первых, это задача разработки и тестирования файловой системы специального назначения, для выполнения которой можно воспользоваться предложенным примером спецификации. Во-вторых, более глубокое изучение зависимости скорости работы алгоритмов компрессии и декомпрессии от технических характеристик компьютера. В-третьих, интеграция разработанного модуля с подсистемой удаленной доставки приложений.

Список литературы

[1] Data Deduplication with Linux [В интернете] / авт. Petros Koutoupis – http://www.linuxjournal.com/content/data-deduplication-linux?page=0,2

[2] Disk Structures [В интернете] / авт. Alex Verstak – http://averstak.tripod.com/fatdox/00dindex.htm

[3] Filesystem Benchmarks - Part I [В интернете] / авт. NeptuneLabs – http://fsi-viewer.blogspot.ru/2011/10/filesystem-benchmarks-part-i.html

[4] Microsoft Extensible Firmware Initiative. FAT32 File System Specification [В интернете] / авт. MicroSoft – http://msdn.microsoft.com/en-us/library/gg463080.aspx

[5] SDFS: Performance Metrics [В интернете] / авт. Opendedup – http://opendedup.org/perfmon

[6] Understanding FAT32 Filesystems [В интернете] / авт. Paul Stoffregen – https://www.pjrc.com/tech/8051/ide/fat32.html

[7] Understanding ZFS: Compression [В интернете] / авт. Ben Rockwood – http://www.cuddletech.com/blog/pivot/entry.php?id=983

[8] ZFS Administration, Appendix D- The True Cost Of Deduplication [В интернете] / авт. Aaron Toponce – https://pthree.org/2013/12/18/zfs-administration-appendix-d-the-true-cost-of-deduplication/

[9] Алгоритмы сжатия [В интернете] / авт. Ливак Е.Н. – http://mf.grsu.by/UchProc/livak/en/po/comprsite/theory_contents.html

[10] Алгоритмы, используемые при сжатии данных [В интернете] / авт. @JagaJaga – http://habrahabr.ru/post/132289/

[11] Введение в дедупликацию данных [В интернете] / авт. Veeam – http://habrahabr.ru/company/veeam/blog/203614/

[12] Дедупликация «на лету» [В интернете] / авт. EMC – http://russia.emc.com/corporate/glossary/inline-deduplication.htm

[13] Дедупликация данных [В интернете] / авт. EMC – http://russia.emc.com/corporate/glossary/data-deduplication.htm

[14] Дедупликация и компрессия: оценка сжатия на реальных данных [В интернете] / авт. Алексей Шипилёв – http://habrahabr.ru/post/104979/

[15] Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео [Книга] / авт. Ватолин Д., Ратушняк А., Смирнов М., Юкин В. - М.: ДИАЛОГ-МИФИ, 2002.

[16] Сжатие информации без потерь [В интернете] / авт. @Singerofthefall – http://habrahabr.ru/post/142242/; http://habrahabr.ru/post/142492/

[17] Файловая система FAT32. Особенности и описание [В интернете] / авт. Эндоатом – http://www.endoatom.com.ua/articles/41-osystems/92-fat32

Приложение 1. Спецификация Fork-FS.

Поскольку данная файловая система основана на FAT32, то все, что верно для FAT32, верно и для Fork-FS, кроме отдельно оговоренных случаев. Ниже приведена спецификация файловой системой с поддержкой файловой операции fork, реализованная на основе спецификации FAT. Использован перевод за авторством Александра Кобеца.

Правила, принятые в этом документе

Числа, которые начинаются с “0x” являются шестнадцатеричными.

Числа без “0x” в начале, являются десятичными.

Фрагменты кода, представленные в документе, написаны на языке программирования ‘C’. Нет строгой типизации и строгого синтаксиса.

Присутствуют несколько фрагментов кода, в которых используются одновременно 16 и 32 битные переменные. Предполагается, что читатель является программистом, и понимает, какие данные не потеряются при преобразовании из 32 в 16-битные значения. Также стоит иметь в виду, что все переменные беззнаковые (UNSIGNED). Не производите вычислений FAT со знаковыми типами переменных, т.к. в некоторых случаях это даст неправильный результат.

Основные особенности (всех типов FAT)

Все файловые системы FAT изначально созданы для компьютеров архитектуры IBM PC. Поэтому особенностью FAT является то, что порядок бит в данных – от старших к младшим (little endian). Будем рассматривать расположенное на диске 32-битное значение FAT как серию из 4-х 8-битных байт – первым будет byte[0] и последним byte[4] – все 32 бита пронумерованы от 00 до 31 (00 самый младший разряд):

```
byte[3]  3 3 2 2 2 2 2 2
         1 0 9 8 7 6 5 4
byte[2]  2 2 2 2 1 1 1 1
         3 2 1 0 9 8 7 6
byte[1]  1 1 1 1 1 1 0 0
         5 4 3 2 1 0 9 8
byte[0]  0 0 0 0 0 0 0 0
         7 6 5 4 3 2 1 0
```

Это имеет значение, потому что когда FAT используется на компьютере с обратным порядком бит (big endian), то данные придётся транслировать между little endian и big endian при каждом чтении и записи на диск.

Файловая система FAT состоит из четырёх основных регионов, расположенных в данном порядке:

0 – Reserved Region

1 – FAT Region

2 – Root Directory Region (doesn’t exist on FAT32 volumes)

3 – File and Directory Data Region

Boot сектор и BPB

Первая важная структура на FAT диске называется BPB (BIOS Parameter Block), она расположена в первом секторе диска, в Reserved Region. Этот сектор ещё иногда называют “boot сектор”, “reserved sector” или “0^{th} sector”, но главное лишь то, что это первый сектор диска.

Первая вещь, которая часто вызывает вопросы. В MS-DOS версии 1.x, не было BPB. В этой первой версии было только два разных формата дисков: первый – односторонний, и второй – двухсторонний 360K 5.25-дюймовый гибкий диск. Определение типа диска происходило по значению из таблицы FAT (младшие 8 бит в FAT[0]).

Этот метод определения типа диска был заменён в MS-DOS версии 2.x путём внесения в boot сектор BPB, а старый метод определения (по первому байту FAT) больше не поддерживается. Все FAT диски должны иметь в boot секторе BPB.

Теперь возникает второй вопрос: Как точно выглядит BPB? BPB boot сектора для MS-DOS 2.x допускает только меньше 65,536 секторов (32 MB делённые на 512-байтные сектора). Это ограничение связано с тем, что поле “total sectors” является 16-битным. Данное ограничение относится и к MS-DOS 3.x, где BPB было модифицировано включением нового 32-битного поля.

Следующее изменение BPB было в операционной системе Microsoft Windows 95 OEM Service Release2 (OSR2), где была представлена FAT32. FAT16 ограничен размером FAT и максимальным количеством кластеров, максимальным размером диска 2 GB на диске с 512-байтными секторами. FAT32 снимает ограничение на максимальный размер диска (раздела) 2 GB.

FAT32 BPB целиком соответствует FAT12/FAT16 BPB, и дополнен полем BPB_TotSec32. Они отличаются, начиная со смещения 36, в зависимости от типа FAT12/FAT16 или FAT32 (описание определения типа FAT будет ниже). Главное, что BPB в boot секторе FAT диска всегда содержит все поля для FAT12/FAT16/FAT32 BPB типов. Таким образом, обеспечена максимальная совместимость FAT дисков, драйверы файловых систем FAT будут правильно определять и поддерживать структуру, потому что она содержит все предопределённые поля.

ЗАМЕТКА: В дальнейшем описании, все поля, начинающиеся с BPB_, являются частью BPB. Все поля, начинающиеся с BS_, являются частью boot сектора (к BPB не принадлежат). Далее показано начало сектора 0 на FAT диске, содержащего BPB:

Boot сектор и BPB структура

Поле	Смещение	Размер (байт)	Описание
BS_jmpBoot	0	3	Jump инструкция на boot code. Это поле имеет две формы: **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90** и **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??** **0x??** значит, что допустимо любое 8-битное значение. Это трёхбайтное поле для Intel x86 команды перехода (jump) на начало загрузочного кода операционной системы. Этот код обычно находится в секторе 0 сразу после BPB, и возможно в других секторах. Допустима любая из приведённых форм. Наиболее часто используется **JmpBoot[0] = 0xEB**.
BS_OEMName	3	8	“MSWIN4.1” Существует много мнений об этом поле. Но это только строка имени. Microsoft ОС не строит ни каких выводов из содержания этого поля. Но некоторые драйверы FAT делают, поэтому есть резон указывать “MSWIN4.1”, для совместимости. Эта строка ещё является косвенным признаком того, что диск форматирован.
BPB_BytsPerSec	11	2	Количество байтов в секторе. Допустимы только эти значения: 512, 1024, 2048 or 4096. Если нужна максимальная совместимость со старыми программами, то должно использоваться только 512. В мире существует много программ, которые жёстко рассчитаны на значение 512. Microsoft ОС корректно поддерживают все допустимые значения. **Заметка:** Не ошибитесь в понимании максимальной совместимости. Если диск отформатирован с физическим размером секторов N, вы должны использовать именно значение N, вплоть до значения 4096. Максимальная совместимость достигается использованием дисков с обычным размером сектора.
BPB_SecPerClus	13	1	Количество секторов в кластере. Значение должно быть числом в степени 2, и больше 0. Разрешённые значения: 1, 2, 4, 8, 16, 32, 64 и 128. Учтите при этом, что произведение “байтов в кластере” (BPB_BytsPerSec*BPB_SecPerClus) должно быть не больше 32K (32*1024). Неправильно считать, что большее значение допустимо. Значения, дающие размер кластера больше 32K не будут правильно работать; не пробуйте их использовать. Некоторые системы допускают размер кластера 64K, но многие установочные программы будут работать не правильно.
BPB_RsvdSecCnt	14	2	Количество секторов в Reserved region (начинается с первого сектора диска). Должно быть больше 0. ДляFAT12 и FAT16 дисков, это значение должно быть только 1. Для FAT32 дисков, обычное значение 32. В мире есть много программ для FAT12 и FAT16, в которые жёстко настроены на значение 1, не проверяя фактическое значение этого поля. Microsoft ОС корректно обрабатывает любое значение.

BPB_NumFATs	16	1	Количество таблиц FAT на диске. Должно быть 2 для любой FAT. Хотя и допустимы и другие значения, многие программы и системы не будут корректно работать. Все Microsoft системы корректно работают с любым значением, тем не менее, рекомендуемое значение 2. Значение 2 даёт избыточность FAT структуры, при этом, в случае потери сектора, данные не потеряются, потому что они дублированы. На не-дисковых носителях (например, карта памяти FLASH), где избыточность не требуется, для экономии памяти может использоваться значение 1, но некоторые драйверы FAT могут работать неправильно.
BPB_RootEntCnt	17	2	Для FAT12 и FAT16 дисков, это поле содержит число 32-байтных элементов корневой директории. Для FAT32 дисков, это поле должно быть 0. Для FAT12 иFAT16 дисков, значение этого поля, умноженное на 32 должно быть кратно BPB_BytsPerSec. Для максимальной совместимости, FAT16 диски должны содержать значение 512.
BPB_TotSec16	19	2	Старое 16-битное поле: общее количество секторов на диске. Это количество включает в себя все четыре региона диска. Значение не 0; но если равно 0, то BPB_TotSec32 должно быть не 0. Для FAT32 дисков, значение всегда 0. Для FAT12 и FAT16 дисков это поле содержит количество секторов, а BPB_TotSec32 равно 0, если значение «умещается» (меньше 0x10000).
BPB_Media	21	1	0xF8 стандартное значение для “жёстких” (не сменных) дисков. Для сменных дисков, обычное значение 0xF0. Разрешённые значения: 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE и 0xFF. Важно, чтобы это же значение было записано в байт таблицы FAT[0]. Это старое правило появилось MS-DOS 1.x для определения типа диска (как рассказано в начале), но сейчас не имеет практического применения.
BPB_FATSz16	22	2	Для FAT12/FAT16 это количество секторов одной FAT. Для FAT32 это значение равно 0, а количество секторов одной FAT содержится в BPB_FATSz32.
BPB_SecPerTrk	24	2	Секторов на дорожке, для interrupt 0x13. Это поле имеет отношение к дискам, имеющим геометрию (состоит из треков, головок и цилиндров) и доступным через interrupt 0x13. Поле содержит геометрическое значение “sectors per track”.
BPB_NumHeads	26	2	Количество головок, для interrupt 0x13. Имеет такое же значение, как и BPB_SecPerTrk. Поле содержит геометрическое значение “count of heads”. Например, у 1.44 MB 3.5-дюймового гибкого диска оно равно 2.
BPB_HiddSec	28	4	Количество скрытых секторов, перед началом данного раздела диска. Это поле имеет отношение только к дискам, доступным по interrupt 0x13. Поле должно содержать 0, если носитель не разбит на разделы. Это поле только для операционной системы.
BPB_TotSec32	32	4	Новое 32-битное поле: общее количество секторов на диске. Это количество включает в себя все четыре региона диска. Может быть 0; если 0, то BPB_TotSec16 должно быть не 0. Для FAT32 дисков, значение всегдане 0. Для FAT12/FAT16 дисков, поле содержит количество секторов, когда BPB_TotSec16 равно 0 (количество равно или больше 0x10000).

С этой отметки, BPB/boot сектор для FAT12 и FAT16 отличаются от BPB/boot сектора для FAT32. Первая таблица показывает структуру FAT12 и FAT16 со смещения 36 boot сектора.

Структура Fat12 и Fat16 со смещения 36

Поле	**Смещение**	**Размер (байт)**	**Описание**
BS_DrvNum	36	1	Int 0x13 номер устройства (например 0x80). Это поле для загрузчика MS-DOS, и устанавливает для INT 0x13номер диска (0x00 для гибких дисков, 0x80 для жёстких дисков). **ЗАМЕТКА:** Это поле только для операционной системы.
BS_Reserved1	37	1	Зарезервировано (используется Windows NT). Форматирующие программы FAT дисков, всегда должны устанавливать 0.
BS_BootSig	38	1	Дополнительная сигнатура (0x29). Байт является индикатором того, что нижеследующие 3 поля присутствуют.
BS_VolID	39	4	Серийный номер диска. Это поле вместе с BS_VolLab позволяет отслеживать смену диска, и отслеживать моменты, когда вставлен другой диск. Этот номер обычно генерируется путём комбинации текущей даты и времени в 32-битное число.
BS_VolLab	43	11	Имя диска. Это имя совпадает с 11-байтным именем, прописанным в корневой директории. **ЗАМЕТКА:** Драйверы FAT должны не забывать обновлять это поле, при изменении имени в корневой директории. Когда имя не задано, поле содержит строку "**`NO NAME`** ".
BS_FilSysType	54	8	Одна из строк "**`FAT12`** ", "**`FAT16`** " или "**`FAT`** " **ЗАМЕТКА:** Многие люди считают, что эта строка для определения типа FAT – FAT12, FAT16 или FAT32 – на диске. Это не так. Заметьте, что это поле вообще не является частью BPB. Это строка только для общей информации, и программы Microsoft вообще не используют это поле для определения типа FAT, потому что оно часто не корректное, или даже отсутствует. Смотрите в этом документе главу Определение типа FAT. Эту строку нужно устанавливать в соответствии с типом FAT, потому что некоторые не-Microsoft драйверы FAT используют это поле.

Структура FAT32 со смещения 36

Поле	Смещение	Размер (байт)	Описание
BPB_FATSz32	36	4	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Это 32-битное поле FAT32 содержит количество секторов одной FAT. При этом BPB_FATSz16 должно быть 0.
BPB_ExtFlags	40	2	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Bits 0-3 – Номер активной FAT, начиная с 0. Актуально, только если выключено зеркалирование. Bits 4-6 – Зарезервировано. Bit 7 = 0 значит FAT зеркалируется на все остальные. Bit 7 = 1 означает, что активна только одна FAT; её номер указывается в битах 0-3. Bits 8-15 – Зарезервировано.
BPB_FSVer	42	2	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. В старшем байте: номер версии. Младший байт – номер промежуточной версии. Это версия FAT32. Это даёт расширять в будущем систему FAT32, и возможность корректно обрабатывать каждую старую версию. Этот документ описывает версию 0:0. Если в этом поле не 0, то предыдущие версии Windows не подключат этот диск. **ЗАМЕТКА:** Дисковые утилиты должны внимательно ориентироваться на версию, и не работать с версией, с которой не совместимы.
BPB_RootClus	44	4	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Номер первого кластера корневой директории. Обычно 2, но может быть и другим. **ЗАМЕТКА:** Дисковые утилиты, которые изменяют положение корневой директории, должны стараться в качестве первого кластера использовать неповреждённый кластер. Специально для того, чтобы в случае аварийного обнуления поля, утилита исправления диска смогла легко найти начало директории.
BPB_FSInfo	48	2	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Номер сектора со структурой FSINFO в зарезервированной части FAT32. Обычно равен 1. **ЗАМЕТКА:** Резервная копия структуры FSINFO располагается в BackupBoot, но только копия, на которую указывается это поле, обновляется постоянно (можно считать основной и этот резервный сектор как один и тот же).
BPB_BkBootSec	50	2	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Если не ноль, то это номер сектора в резервной области диска, где хранится копия boot сектора. Обычно = 6. Другие значения не рекомендуются.
BPB_Reserved	52	12	Это поле есть только в FAT32 и отсутствует в FAT12 и FAT16. Зарезервировано. В FAT32 все байты этого поля должны быть выставлены в 0.
BS_DrvNum	64	1	Это поле такое же, как в FAT12 и FAT16. Отличие только в том, что у него другое смещение.

BS_Reserved1	65	1	Это поле такое же, как в FAT12 и FAT16. Отличие только в том, что у него другое смещение.
BS_BootSig	66	1	Это поле такое же, как в FAT12 и FAT16. Отличие только в том, что у него другое смещение.
BS_VolID	67	4	Это поле такое же, как в FAT12 и FAT16. Отличие только в том, что у него другое смещение.
BS_VolLab	71	11	Это поле такое же, как в FAT12 и FAT16. Отличие только в том, что у него другое смещение.
BS_FilSysType	82	8	Для Fork-FS в этой строке всегда "**FORK** ". В случае FAT32 – "**FAT32** ". Смотрите описание этого поля в FAT12/FAT16. Нельзя использовать для определения типа FAT. Используется для определения модификации FAT32 – Fork-FS

Нет ничего специфичного в Секторе 0 диска FAT. Рассматривая сектор как массив байт, sector[510] содержит 0x55, а sector[511] содержит 0xAA.

ЗАМЕТКА: Во многих описаниях FAT ошибочно говорится, что сигнатура 0xAA55 расположена в "последних 2 байтах boot сектора". Это верно, если – и только если – BPB_BytsPerSec равняется 512. Если BPB_BytsPerSec больше чем 512, смещение этой сигнатуры не изменяется (хотя допустимо, что и в конце сектора эта сигнатура тоже будет присутствовать).

Обратите внимание на поле BPB_TotSec16/32. Предположим, мы имеем диск или раздел с количеством секторов DskSz. Если поле BPB TotSec (BPB_TotSec16 или BPB_TotSec32 – которое не нулевое) *меньше или равно* DskSz, то ни какой ошибки в FAT нет. Ни чего страшного, если значение BPB_TotSec16/32 немного меньше DskSz. Так же, совершенно нормально, если BPB_TotSec16/32 будет значительно меньше DskSz. Это означает лишь, что часть дискового пространства не используется. Это не означает, что диск FAT повреждён. Однако, если BPB_TotSec16/32 *больше* чем DskSz, то диск серьёзно повреждён или искажён, потому что он расширен за границу носителя или перекрывает данные следующего раздела. Использование диска, у которого BPB_TotSec16/32 "слишком большое" для носителя или данного раздела, может повлечь катастрофичную потерю данных.

Структура FAT

Теперь рассмотрим саму структуру FAT. Структура содержит однонаправленные списки "блоков" (кластеров) файлов. Директория (или контейнер файлов) FAT представляет собой не что иное, а обычный файл, со специальным атрибутом «директория». Особенностью директории является то, что хранимые данные (файлы) представляют собой массив 32-байтных структур (их описание ниже). Во всём остальном, директория ни чем не отличается от файла.

Для реализации fork() FAT модифицируется, путем добавления дополнительного типа структур «wrap», выполненного на основе директорий. Данная структура отличается значением специального атрибута «оболочка». Назначением данной структуры является хранение в виде 32-битных структур «ссылок» на оригинальный файл и на группы цепочек примененных к нему изменений.

FAT распределяет данные по номерам кластеров. Номер самого первого кластера 2.

Первый сектор кластера 2 (в регионе данных) вычисляется через поля BPB как описано ниже. Сначала определяем количество секторов, занятых корневой директорией:

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;
```

Заметьте, что в FAT32 поле BPB_RootEntCnt всегда содержит 0, поэтому для FAT32 RootDirSectors всегда 0. Здесь число 32 это размер одного элемента директории FAT (в байтах). Также заметьте, что округление производится *вверх*.

Начало региона данных, первый сектор кластера 2, вычисляется так:

```
If(BPB_FATSz16 != 0)
        FATSz = BPB_FATSz16;
Else
        FATSz = BPB_FATSz32;

FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;
```

ЗАМЕТКА: Этот номер сектора является относительным номеру сектора, содержащим BPB (считаем номер сектора BPB равным 0). Не считайте, что это номер от начала носителя, потому что сектор 0 диска не обязательно является сектором 0 носителя, делённого на разделы.

Имея номер кластера **N**, номер первого сектора этого кластера (опять же относительно сектора 0 диска) вычисляется так:

```
FirstSectorofCluster = ((N – 2) * BPB_SecPerClus) + FirstDataSector;
```

ЗАМЕТКА: Поскольку значение BPB_SecPerClus ограничено степенью 2 (1,2,4,8,16,32….), значит, умножения и деления над BPB_SecPerClus можно производить операцией двоичного сдвига SHIFT на процессорах, где эта операция быстрей операций MULT и DIV. На современных Intel X86 процессорах, это не имеет большого смысла, поскольку машинные инструкции MULT и DIV сильно оптимизированы для операций над значениями в степени 2.

Определение типа FAT

Существует много ошибочных мнений, как именно это производится, и возникают погрешности “на 1”, “на 2”, “на 10” и “очень большие”. На самом деле принцип простой. Тип FAT – один из FAT12,FAT16 или FAT32 – определяется по количеству кластеров на диске, и *ни чем* иным. Для определения Fork-FS необходимо удостовериться, что файловая система по количеству кластеров попадает под определение FAT32, и BS_FilSysType содержит строку “FORK”.

Читайте внимательно, здесь много тонких моментов. Например, “число кластеров”. Это не то же самое, что и “максимальный номер кластера”, потому что кластеры начинают считаться с 2, а не 0 или 1.

Для начала, давайте посмотрим, как именно определяется “число кластеров”. Оно определяется из полей BPB диска. Сначала, определяем количество секторов, занимаемых корневой директорией, как было показано выше.

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;
```

Заметьте, что в FAT32 поле BPB_RootEntCnt всегда содержит 0, поэтому для FAT32 RootDirSectors всегда 0.

Затем, определяем количество секторов в регионе данных диска:

```
If(BPB_FATSz16 != 0)
        FATSz = BPB_FATSz16;
Else
        FATSz = BPB_FATSz32;

If(BPB_TotSec16 != 0)
        TotSec = BPB_TotSec16;
Else
        TotSec = BPB_TotSec32;

DataSec = TotSec – (BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
```

Теперь определяем количество кластеров:

```
CountofClusters = DataSec / BPB_SecPerClus;
```

Заметьте, что это вычисление округляет *вниз*.

Теперь мы можем определить тип FAT. *Пожалуйста, читайте внимательно, чтобы не получить ошибку погрешности на 1!*

К примеру, когда мы говорим <, это не означает <=. Также будьте уверены, что все числа правильные. Первое число для FAT12 это 4085; второе число для FAT16 это 65525. Эти значения и знак ‘<’ не ошибочны.

```
If(CountofClusters < 4085) {
        /* Volume is FAT12 */
} else if(CountofClusters < 65525) {
        /* Volume is FAT16 */
else if(BS_FilSysType == “FORK”) {
        /* Volume is Fork-FS */
} else {
        /* Volume is FAT32 */
}
```

Это единственный способ определения типа FAT. Не бывает дисков FAT12, у которых кластеров больше 4 084. Не бывает дисков FAT16, у которых кластеров меньше 4 085 или больше 65 524. Не бывает дисков FAT32, у которых кластеров меньше 65 525. Если вы создадите диск FAT, который не соответствует этим правилам, то Microsoft системы будут с ними работать не правильно, т.к. станут считать что это FAT другого типа, чем считаете Вы.

ЗАМЕТКА: Как уже несколько раз было сказано, мир наполнен большим количеством ошибочного кода для FAT. Много кода с погрешностью 1, 2, 8, 10 или 16. Поэтому очень рекомендуется форматировать диски FAT с учетом максимальной совместимости с существующим кодом, и стараться избегать создания дисков с количеством кластеров, близким к 4 085 или 65 525. Используйте значения кластеров на 16 больше или меньше от этих граничных чисел.

Заметьте также, что значение CountofClusters – именно *количество* кластеров данных, которые начинаются с номера 2. Максимальный номер кластера на диске равен CountofClusters + 1, а “количество кластеров, включая 2 резервных” равно CountofClusters + 2.

Покажем ещё одно важное вычисление. Имея номер кластера **N**, как найти входную точку в таблице FAT? Из всех типов FAT, только FAT12 представляет сложность. Для FAT16 и FAT32 вычисления простые:

```
If(BPB_FATSz16 != 0)
        FATSz = BPB_FATSz16;
Else
        FATSz = BPB_FATSz32;

If(FATType == FAT16)
        FATOffset = N * 2;
Else if (FATType == FAT32)
        FATOffset = N * 4;

ThisFATSecNum = BPB_ResvdSecCnt + (FATOffset / BPB_BytsPerSec);
ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

REM(…) это оператор остатка от деления. Так получаем остаток от деления FATOffset на BPB_BytsPerSec. ThisFATSecNum это номер сектора, содержащий входную точку кластера N в первой таблице FAT. Если вы хотите получить номер сектора во второй FAT, вам нужно прибавить FATSz к ThisFATSecNum; для третьей FAT, вы прибавите 2*FATSz, и так далее.

Выполните чтение сектора номер ThisFATSecNum (помните, что это номер относительно сектора 0 диска). Считаем, что прочитали в массив 8-битных байтов SecBuff. Также считаем, что WORD является 16-битным беззнаковым, а DWORD 32-битным беззнаковым.

```
If(FATType == FAT16)
        FAT16ClusEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
Else
        FAT32ClusEntryVal = (*((DWORD *) &SecBuff[ThisFATEntOffset])) & 0x0FFFFFFF;
```

Прочитали значение кластера. Запись значения в тот же кластер делается так:

```
If(FATType == FAT16)
        *((WORD *) &SecBuff[ThisFATEntOffset]) = FAT16ClusEntryVal;
Else {
        FAT32ClusEntryVal = FAT32ClusEntryVal & 0x0FFFFFFF;
        *((DWORD *) &SecBuff[ThisFATEntOffset]) =
                (*((DWORD *) &SecBuff[ThisFATEntOffset])) & 0xF0000000;
        *((DWORD *) &SecBuff[ThisFATEntOffset]) =
                (*((DWORD *) &SecBuff[ThisFATEntOffset])) | FAT32ClusEntryVal;
}
```

Поясним работу кода для FAT32. У FAT32, значения FAT занимают только 28 бит, а старшие 4 бита зарезервированы. Данные в старшие 4 бита FAT32 значений FAT записываются только при форматировании диска, при этом все биты просто обнуляются.

Здесь дадим ещё немного пояснений о значениях FAT для FAT32. В общем, 32-битные FAT значения содержат не совсем 32-битные значения; из них используются только 28 бит. К примеру, все эти 32-битные номера кластеров: 0x10000000, 0xF0000000 и 0x00000000 являются свободными (FREE), потому что

старшие 4 бита должны игнорироваться. Если 32-битный свободный кластер содержит значение 0x30000000, и Вы хотите пометить его как «BAD CLUSTER» значением 0x0FFFFFF7, то в результате 32-битное значение получится 0x3FFFFFF7, потому что Вы должны оставить неизменным значение в старших 4 битах, записывая значение 0x0FFFFFF7 «BAD CLUSTER».

Заметим, что поскольку значение BPB_BytsPerSec всегда кратно 2 и 4, то для FAT16 и FAT32 значений FAT можно не беспокоиться об их расположении на границе между секторами (но не для FAT12).

Код для FAT12 более сложный, потому что здесь 1,5 байт (12-бит) на каждое значение FAT.

```
if (FATType == FAT12)
        FATOffset = N + (N / 2);
/* Multiply by 1.5 without using floating point, the divide by 2 rounds DOWN */
        ThisFATSecNum = BPB_ResvdSecCnt + (FATOffset / BPB_BytsPerSec);
        ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

Теперь нужно проверить пересечение значением границы сектора:

```
If(ThisFATEntOffset == (BPB_BytsPerSec - 1)) {
       /* This cluster access spans a sector boundary in the FAT */
       /* There are a number of strategies to handling this. The */
       /* easiest is to always load FAT sectors into memory */
       /* in pairs if the volume is FAT12 (if you want to load */
       /* FAT sector N, you also load FAT sector N+1 immediately */
       /* following it in memory unless sector N is the last FAT */
       /* sector). It is assumed that this is the strategy used here */
       /* which makes this if test for a sector boundary span */
       /* unnecessary. */
}
```

Обращаемся к значению FAT как к WORD (также как в FAT16), но если номер кластера ЧЁТНЫЙ, мы используем только младшие 12 бит из 16-битного прочитанного значения; и если номер кластера НЕЧЁТНЫЙ, мы используем только старшие 12 бит.

```
FAT12ClusEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
If(N & 0x0001)
        FAT12ClusEntryVal = FAT12ClusEntryVal >> 4; /* Cluster number is ODD */
Else
        FAT12ClusEntryVal = FAT12ClusEntryVal & 0x0FFF; /* Cluster number is EVEN */
```

Прочитали значение кластера. Запись значения в тот же кластер делается так:

```
If(N & 0x0001) {
        FAT12ClusEntryVal = FAT12ClusEntryVal << 4; /* Cluster number is ODD */
        *((WORD *) &SecBuff[ThisFATEntOffset]) =
                (*((WORD *) &SecBuff[ThisFATEntOffset])) & 0x000F;
} Else {
        FAT12ClusEntryVal = FAT12ClusEntryVal & 0x0FFF; /* Cluster number is EVEN */
        *((WORD *) &SecBuff[ThisFATEntOffset]) =
                (*((WORD *) &SecBuff[ThisFATEntOffset])) & 0xF000;
}
*((WORD *) &SecBuff[ThisFATEntOffset]) =
        (*((WORD *) &SecBuff[ThisFATEntOffset])) | FAT12ClusEntryVal;
```

ЗАМЕТКА: Полагаем, что оператор >> заполнит значением 0 старшие 4 бита, а оператор << заполнит значением 0 младшие 4 бита.

Данные файла ассоциируются с файлом следующим образом. В директории содержится номер первого кластера, в котором располагаются данные файла. Данные первого кластера ассоциированы с номером первого кластера, и расположение данных вычисляется из номера первого кластера, как было показано выше (вычисление FirstSectorofCluster).

В Fork-FS предлагается каждый файл хранить в отдельной структуре “wrap”, выполненной на основе директорий. Данная структура в общем случае содержит номер первого кластера, в котором располагаются данные файла, либо номер первого кластера, содержащего исходный файл, в случае образования нового файла в результате операции fork. Также структура содержит номера кластеров начала/конца цепочек изменений исходного файла.

Файлы нулевой длинны (файлы, не содержащие данных) имеют в директории номер первого кластера 0. Кластер, расположенный в FAT (смотри вычисление ThisFATSecNum и ThisFATEntOffset) может содержать значение EOC (End Of Clusterchain – конец цепочки кластеров) или номер следующего кластера файла. Значение EOC зависит от типа FAT (здесь FATContent является значением кластера прочитанного из FAT, для проверки на значение EOC):

```
IsEOF = FALSE;
If(FATType == FAT12) {
        If(FATContent >= 0x0FF8)
                IsEOF = TRUE;
} else if(FATType == FAT16) {
        If(FATContent >= 0xFFF8)
                IsEOF = TRUE;
} else if (FATType == FAT32) {
        If(FATContent >= 0x0FFFFFF8)
                IsEOF = TRUE;
}
```

Номер кластера, по которому в таблице FAT содержится значение EOC, содержит данные файла, и является последним кластером файла или последним кластером цепочки изменений. При этом, поскольку цепочка изменений может являться вставкой посреди исходного файла, то наряду с номером первого кластера цепочки в структуре “wrap” хранится и номер кластера, следующего за цепочкой изменений.

Microsoft системные драйвера FAT используют такие EOCзначения: 0x0FFF для FAT12, 0xFFFF для FAT16, 0x0FFFFFFF для FAT32. Однако существуют дисковые утилиты для Microsoft ОС, которые используют другие значения.

Опишем специальное значение “BAD CLUSTER”. Все кластеры, содержащие значение “BADCLUSTER” в FAT, не должны присутствовать в списке свободных кластеров, т.к. это вызовет ошибку работы с диском. Значения “BAD CLUSTER”: 0x0FF7 для FAT12, 0xFFF7 для FAT16, 0x0FFFFFF7 дляFAT32. Эти “BAD CLUSTER” похожи на потерянные кластеры, и выглядят как распределённые, потому что содержат ненулевые данные. Дисковые утилиты должны считать потерянные кластеры со специальным значением “BAD CLUSTER” не доступными, и не должны изменять их данные.

ЗАМЕТКА: Значение “BAD CLUSTER” не может входить в диапазон доступных кластеров на дискахFAT12 и FAT16, но значение 0x0FFFFFF7 может быть доступным кластером на дисках FAT32. Для исключения конфликтов с дисковыми утилитами, на дисках FAT32 кластер 0x0FFFFFF7 использоваться не должен.

Список свободных кластеров FAT это просто все кластеры, которые содержат значение 0 в таблице FAT. Эти значения считываются точно так же, было описано выше для ненулевых значений. Список свободных кластеров нигде отдельно не хранится; он должен вычисляться каждый раз при подключении диска к системе, путём сканирования в таблице FAT всех значений 0. На дисках FAT32, в секторе BPB_FSInfo *может* содержаться актуальное количество свободных кластеров. Смотрите документацию на сектор FAT32 FSInfo.

Что содержат первые два зарезервированные кластера в таблице FAT? Первый зарезервированный кластер, FAT[0], содержит значение BPB_Media в младших 8 битах, а все остальные биты содержат 1. К примеру, если BPB_Media значение 0xF8, то для FAT12 FAT[0] = 0x0FF8, для FAT16 FAT[0] = 0xFFF8, а для FAT32 FAT[0] = 0x0FFFFFF8. Во второй зарезервированный кластер, FAT[1], FORMAT устанавливает значение EOC. На дисках FAT12 он не используется, потому что содержит только лишь значение EOC. На дисках FAT16 и FAT32, драйвер файловой системы может использовать старшие два бита в FAT[1] для индикаторов «корректности» (остальные биты содержат 1). Заметьте, что расположение битов для FAT16 и FAT32 разное, потому что они занимают старшие 2 бита.

Для FAT16:

```
ClnShutBitMask = 0x8000;
HrdErrBitMask = 0x4000;
```

Для FAT32:

```
ClnShutBitMask = 0x08000000;
HrdErrBitMask = 0x04000000;
```

Бит ClnShutBitMask – если 1, диск “чист”; если 0, диск “грязен”. Это признак того, что диск не был корректно отключен (Dismount) после последнего подключения. Это хороший повод для запуска для него утилиты проверки и исправления Chkdsk/Scandisk, потому что он возможно повреждён.

Бит HrdErrBitMask – если 1, ошибок чтения/записи не было; если 0, в драйвере файловой системы произошла ошибка чтения/записи диска, что является индикатором того, что некоторые секторы испортились. Это хороший повод для запуска утилиты Chkdsk/Scandisk для проверки поверхности диска и поиска испорченных секторов.

Два важных замечания о регионе FAT на диске:

1. Последний сектор FAT не обязательно весь занят таблицей FAT. В последнем секторе, FAT заканчивается на номере последнего кластера FAT, который равен CountofClusters + 1 (смотри вычисление CountofClusters выше), и это значение FAT не обязательно расположено в конце последнего сектора таблицы FAT. FAT драйвер не должен предполагать, что остаток последнего FAT сектора, после кластера CountofClusters + 1 заполнен определёнными значениями. Программа FAT format остаток сектора должна обнулять.

2. BPB_FATSz16 (BPB_FATSz32 для FAT32) значение *может* быть больше, чем требуется. Другими словами, здесь могут быть полностью неиспользуемые секторы FAT в конце каждой таблицы FAT в регионе FAT диска. Поэтому, последний сектор таблицы FAT всегда вычисляется от значения CountofClusters + 1, а не от BPB_FATSz16/32. FAT драйвер не должен предполагать, что «лишние» FAT сектора заполнены определёнными значениями. Программа FAT format «лишние» сектора должна обнулять.

Разметка нового FAT диска

К этому моменту, у внимательного читателя возникнет интересный вопрос. Тип FAT (FAT12, FAT16,or FAT32) зависит от количества кластеров – а количество секторов региона данных зависит от размера таблицы FAT – когда имеется неформатированный диск без BPB, как всё это вычислить, и записать правильные значения в BPB_SecPerClus, BPB_FATSz16 (или BPB_FATSz32)? Microsoft ОС делает вычисления, используя фиксированные значения, некоторые таблицы и немного арифметики.

Microsoft ОС может создать FAT12 только на гибких дисках. Поскольку форматов гибких дисков существует ограниченное количество, и они имеют фиксированный размер, то считается так:

```
"Если гибкий диск такого типа, то BPB выглядит так."
```

Не делается динамических вычислений FAT12. Для FAT12, все вычисления для BPB_SecPerClus и BPB_FATSz16 делаются на бумаге, и вносятся в таблицы (внимательно следя, чтобы количество кластеров было меньше 4085). Если носитель размером больше 4 MB, не используйте FAT12. Используйте меньшие значения BPB_SecPerClus, чтобы получился FAT16.

Остаток этой главы целиком специфичен для дисков с размером сектора 512 байт. Вы не можете использовать эти таблицы и вычисления для дисков, имеющих другой размер сектора. “Фиксированные значения” это просто размеры дисков “разделяющие FAT16 и FAT32”. Все значения, которые меньше, являются дисками FAT16, и все значения равные или больше, являются дисками FAT32. В Windows, это значение 512 MB. Любые FAT диски меньше 512 MB являются FAT16, и любые FAT диски размером 512 MB или больше, являются FAT32.

Пожалуйста, не сделайте ложных выводов. Многие диски FAT16 имеют размер больше 512 MB. Есть возможность использования формата FAT16 вместо FAT32, программы могут не использовать этот лимит. В данном случае речь идёт о формате *по умолчанию* для MS-DOS и Windows дисков, которые ещё не форматированы. Здесь две таблицы – одна для FAT16 и другая для FAT32. Значениями в этих таблицах являются размеры дисков в 512 байтных секторах (значения для полей BPB_TotSec16 или BPB_TotSec32), а производные значения являются BPB_SecPerClus.

```
struct DSKSZTOSECPERCLUS {
         DWORD DiskSize;
         BYTE SecPerClusVal;
};

/*
        *This is the table for FAT16 drives. NOTE that this table includes
        * entries for disk sizes larger than 512 MB even though typically
        * only the entries for disks < 512 MB in size are used.
        * The way this table is accessed is to look for the first entry
        * in the table for which the disk size is less than or equal
        * to the DiskSize field in that table entry. For this table to
        * work properly BPB_RsvdSecCnt must be 1, BPB_NumFATs
        * must be 2, and BPB_RootEntCnt must be 512. Any of these values
        * being different may require the first table entries DiskSize value
        * to be changed otherwise the cluster count may be to low for FAT16.
*/
DSKSZTOSECPERCLUS DskTableFAT16 [] = {
        { 8400, 0}, /* disks up to 4.1 MB, the 0 value for SecPerClusVal trips an error */
        { 32680, 2}, /* disks up to 16 MB, 1k cluster */
        { 262144, 4}, /* disks up to 128 MB, 2k cluster */
        { 524288, 8}, /* disks up to 256 MB, 4k cluster */
        { 1048576, 16}, /* disks up to 512 MB, 8k cluster */
        /* The entries after this point are not used unless FAT16 is forced */
        { 2097152, 32}, /* disks up to 1 GB, 16k cluster */
        { 4194304, 64}, /* disks up to 2 GB, 32k cluster */
        { 0xFFFFFFFF, 0} /* any disk greater than 2GB, 0 value for SecPerClusVal trips an error */
};

/*
        * This is the table for FAT32 drives. NOTE that this table includes
        * entries for disk sizes smaller than 512 MB even though typically
        * only the entries for disks >= 512 MB in size are used.
        * The way this table is accessed is to look for the first entry
        * in the table for which the disk size is less than or equal
        * to the DiskSize field in that table entry. For this table to
        * work properly BPB_RsvdSecCnt must be 32, and BPB_NumFATs
        * must be 2. Any of these values being different may require the first
        * table entries DiskSize value to be changed otherwise the cluster count
        * may be to low for FAT32.
*/
        DSKSZTOSECPERCLUS DskTableFAT32 [] = {
        { 66600, 0}, /* disks up to 32.5 MB, the 0 value for SecPerClusVal trips an error */
        { 532480, 1}, /* disks up to 260 MB, .5k cluster */
        { 16777216, 8}, /* disks up to 8 GB, 4k cluster */
        { 33554432, 16}, /* disks up to 16 GB, 8k cluster */
        { 67108864, 32}, /* disks up to 32 GB, 16k cluster */
        { 0xFFFFFFFF, 64}/* disks greater than 32GB, 32k cluster */
};
```

Так для данного размера диска FAT (FAT16 или FAT32), мы теперь имеем значение BPB_SecPerClus. Только одну вещь нам осталось вычислить – количество секторов для таблицы FAT (BPB_FATSz16 или

BPB_FATSz32). Считаем, что поля BPB_RootEntCnt, BPB_RsvdSecCnt и BPB_NumFATs уже корректно установлены. Так же считаем, что DskSize содержит размер диска, который будем записывать в BPB_TotSec32 или BPB_TotSec16.

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;
TmpVal1 = DskSize – (BPB_ResvdSecCnt + RootDirSectors);
TmpVal2 = (256 * BPB_SecPerClus) + BPB_NumFATs;
If(FATType == FAT32)
         TmpVal2 = TmpVal2 / 2;
FATSz = (TMPVal1 + (TmpVal2 – 1)) / TmpVal2;
If(FATType == FAT32) {
         BPB_FATSz16 = 0;
         BPB_FATSz32 = FATSz;
} else {
       BPB_FATSz16 = LOWORD(FATSz);
       /* there is no BPB_FATSz32 in a FAT16 BPB */
}
```

Не тратьте много времени на разбор данного кода. Базис этих вычислений сложен; мы просто показали, как это делает Microsoft ОС, и как это работает. Однако заметим, что в приведённом коде имеются недостатки. В некоторых случаях для FAT16 FATSz получается на 2 сектора больше чем нужно, а также для FAT32 иногда получается на 8 секторов больше необходимого. Главное, что FATSz не бывает меньше необходимого. Поскольку допустимо иметь FATSz больше необходимого, то ценой потери нескольких секторов мы получаем запас, который в некоторых случаях может пригодиться.

Структура сектора FAT32 FSInfo и сектора Backup Boot

На диске FAT32, таблица FAT может иметь значительный размер, в отличие от FAT16, ограниченного размером 128K, и FAT12, ограниченного размером 6K. Поэтому есть резон хранить “последнее известное” количество свободных кластеров диска FAT32, чтобы не вычислять его каждый раз, когда функции API запрашивают размер свободного места на диске (например, в конце вывода содержимого директории). Номер сектора FSInfo хранится в поле BPB_FSInfo; в Microsoft ОС оно всегда равно 1. Это структура сектора FSInfo:

FAT32 FSInfo Sector Structure and Backup Boot сектор

Поле	**Сме-щение**	**Размер (байт)**	**Описание**
FSI_LeadSig	0	4	Значение 0x41615252. Это начальная сигнатура для точного определения сектора FSInfo.
FSI_Reserved1	4	480	Зарезервировано на будущее. Программа FAT32 format должна заполнять его 0. Байты этого поля сейчас не используются.
FSI_StrucSig	484	4	Значение 0x61417272. Эта сигнатура для точного определения положения следующих за ним полей.

FSI_Free_Count	488	4	Хранит “последнее известное” количество свободных кластеров диска. Если равно 0xFFFFFFFF, то количество неизвестно, и должно быть вычислено. Может быть любое другое значение, при этом не обязательно корректное. Оно должно проверяться на выполнение условия <= количества кластеров на диске.
FSI_Nxt_Free	492	4	Вспомогательное значение для драйвера FAT. Содержит номер кластера, начиная с которого надо искать свободный кластер. Поскольку FAT32 таблица FAT велика, могут происходить значительные затраты времени, когда в начале FAT много занятых кластеров, а драйвер ищет свободный начиная с номера 2. Обычно здесь номер последнего выделенного кластера. Если здесь значение 0xFFFFFFFF, то нет вспомогательного значения, и поиск должен производиться с номера 2. Любое другое значение может использоваться, но должно предварительно проверяться на условие <= количества кластеров на диске.
FSI_Reserved2	496	12	Зарезервировано на будущее. Программа FAT32 format должна заполнять его 0. Байты этого поля сейчас не используются.
FSI_TrailSig	508	4	Значение 0xAA550000. Это конечная сигнатура для точного определения сектора FSInfo. Заметьте, что старшие 2 байта этого значения (которые располагаются в байтах 510 и 511) соответствуют сигнатуре с таким же смещением в секторе 0.

FAT32 дополнен полем, которого нет в FAT16/FAT12 – BPB_BkBootSec. Диски FAT16/FAT12 могут быть полностью потеряны, если содержимое сектора 0 затрётся, или повредится и перестанет читаться. Это “смертельная точка” для FAT16 и FAT12 дисков. Поле BPB_BkBootSec уменьшает эту уязвимость для дисков FAT32, потому что, начиная с сектора номер 6, имеется резервная копия boot сектора, включая BPB.

В случае если сектор 0 ошибочно затрётся, любая утилита исправления дисков просто восстановит boot сектор из резервной копии.

Для второго случая – где сектор 0 повреждается – есть резон использовать для BPB_BkBootSec именно значение 6. Если сектор 0 не читаемый, некоторые ОС “жёстко настроены” на проверку резервного boot сектора начиная с номера 6 на диске FAT32. Заметьте, что, начиная с сектора BPB_BkBootSec, находится *полная* boot копия. Microsoft FAT32 “boot сектор” состоит из трёх 512-байтных секторов. Здесь копия всех трёх секторов, в секторе начиная с BPB_BkBootSec. Копия сектора FSInfo тоже здесь, хотя поле BPB_FSInfo в резервном boot секторе содержит такое же значение, как и в секторе 0 BPB.

ЗАМЕТКА: Все эти 3 сектора имеют сигнатуру 0xAA55 на смещении 510 и 511, так же как первый boot сектор (смотри описание выше, в конце описания структуры BPB).

Структура Директории FAT

Пока мы будет говорить о записях с короткими именами директорий, не беря во внимание записей длинных имен.

Директория FAT – ни что иное как “файл”, состоящий из списка 32-байтных структур. Лишь одна специальная директория, которая всегда должна присутствовать, это корневая директория. На дисках FAT12 и FAT16, корневая директория расположена в фиксированном месте – непосредственно после последней таблицы FAT, и состоит из фиксированного количества секторов, вычисляемого из BPB_RootEntCnt (смотри

вычисление RootDirSectors выше). Для дисков FAT12 и FAT16, номер первого сектора корневой директории зависит от номера первого сектора таблицы FAT:

```
FirstRootDirSecNum = BPB_ResvdSecCnt + (BPB_NumFATs * BPB_FATSz16);
```

Для FAT32, корневая директория может быть произвольного размера из последовательности кластеров, так же как любая другая директория. Номер первого кластера корневой директории FAT32 хранится в BPB_RootClus. В отличие от других директорий, корневая директория любой FAT не имеет штампа даты и времени, не имеет имени файла (кроме неявного имени "\"), и не содержит "." и ".." в первых двух записях. Ещё один аспект – только в корневой директории может содержаться файл, у которого установлен единственный атрибут ATTR_VOLUME_ID (смотри ниже).

FAT 32 Byte Directory Entry Structure

Поле	**Сме-щение**	**Размер (байт)**	**Описание**
DIR_Name	0	11	Короткое имя.
DIR_Attr	11	1	Атрибуты файла: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_WRAP 0x20 ATTR_LINK 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID
DIR_NTRes	12	1	Зарезервировано и устанавливается в 0.
DIR_CrtTimeTenth	13	1	Штамп миллисекунд текущего времени. Фактически это поле содержит десятые доли секунды. Гранулярность секунд DIR_CrtTime равна 2 секундам, поэтому диапазон допустимых значения значений 0-199 включительно.
DIR_CrtTime	14	2	Время создания файла.
DIR_CrtDate	16	2	Дата создания файла.
DIR_LstAccDate	18	2	Дата последнего обращения. Заметим, что здесь нет времени последнего обращения, только дата. Это дата последнего чтения или записи. В случае записи, дата должна быть такой же, как DIR_WrtDate.
DIR_FstClusHI	20	2	Старшее слово номера первого кластера (всегда 0 для FAT12 и FAT16).
DIR_WrtTime	22	2	Время последней записи. Заметим, что создание файла считается записью.
DIR_WrtDate	24	2	Дата последней записи. Заметим, что создание файла считается записью.
DIR_FstClusLO	26	2	Младшее слово номера первого кластера.
DIR_FileSize	28	4	32-битный DWORD содержит размер файла в байтах.

DIR_Name[0]

Специальное назначение первого байта (DIR_Name[0]) в записи директории FAT:

- Если DIR_Name[0] == 0xE5, то запись директории свободна (в этой записи нет ни файла, ни директории).
- Если DIR_Name[0] == 0x00, то запись директории свободна (как и 0xE5), и нет существующих записей после этой (все DIR_Name[0] байты после этой записи также равны 0). Специальное значение 0 (но не значение 0xE5) индицирует драйверу FAT, что оставшиеся записи директорий можно не анализировать, потому что они все свободны.
- Если DIR_Name[0] == 0x05, то настоящий байт имени файла равен 0xE5. 0xE5 соответствует старшему байту японской кодировки KANJI. Специальное значение 0x05 использовано потому, что данное значение можно корректно интерпретировать и в японской кодировке, и драйвер FAT не будет считать запись свободной.

Поле DIR_Name состоит из двух частей, 8-символьная основная часть имени, и 3-сомвольное расширение. Свободные концы обеих частей заполнены байтами 0x20.

DIR_Name[0] не может быть равным 0x20. Между основной частью и расширением подразумевается символ ‘.’, который отсутствует в DIR_Name. Нижний регистр букв в DIR_Name не разрешён (это специфично для разных стран).

Следующие символы не разрешены в DIR_Name:

- Значения меньше 0x20, за исключением 0x05 в DIR_Name[0] описанного выше.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, 0x7C.

Несколько примеров записи имён в DIR_Name:

```
“foo.bar” -> “FOO BAR”
“FOO.BAR” -> “FOO BAR”
“Foo.Bar” -> “FOO BAR”
“foo” -> “FOO ”
“foo.” -> “FOO ”
“PICKLE.A” -> “PICKLE A ”
“prettybg.big” -> “PRETTYBGBIG”
“.big” -> illegal, DIR_Name[0] cannot be 0x20
```

В директориях FAT все имена уникальны. Посмотрите на первые три примера выше. Эти три имени соответствуют одному и тому же файлу, а в директории может быть только одно DIR_Name с именем “FOO BAR”.

DIR_Attr содержит атрибуты файла:

ATTR_READ_ONLY	Означает что разрешено только чтение, а запись запрещена.
ATTR_HIDDEN	Означает что в обычном списке файлов он не будет показан.
ATTR_SYSTEM	Означает что это файл операционной системы.
ATTR_VOLUME_ID	На диске должен быть только один “файл” с этим атрибутом, и располагаться в корневой директории. Имя этого файла фактически является ярлыком диска. DIR_FstClusHI и DIR_FstClusLO для ярлыка диска должны быть всегда 0 (файл не содержит ни каких данных).
ATTR_DIRECTORY	Означает что фактически файл содержит список других файлов.
ATTR_ARCHIVE	Атрибут используется утилитами резервного копирования. Этот бит устанавливается в 1 драйвером FAT когда файл создаётся,

	переименовывается, или записываются данные. Для утилит резервного копирования он означает, что файл изменился с момента последнего резервного копирования.
ATTR_WRAP	Означает что файл является «оболочкой» и содержит в себе некоторый файл (или ссылку на него) и список примененных к нему изменений.
ATTR_LINK	Означает что файл является «ссылкой» на некоторый исходный файл. Данный тип файла может быть получен в результате выполнения файловой операции fork, когда создается файл-оболочка, первой записью которого является ссылка на исходный файл.

Отметим, что комбинация атрибутов ATTR_LONG_NAME означает, что “файл” является лишь частью длинного имени другого файла или же используется как счетчик в Fork-FS. Описание этой комбинации атрибутов дано в соответствующем параграфе.

Когда создаётся директория (файл с установленным атрибутом ATTR_DIRECTORY в поле DIR_Attr), DIR_FileSize устанавливается в 0. DIR_FileSize не используется, оно всегда равно 0 у файлов с атрибутом ATTR_DIRECTORY (директория состоит просто из списка кластеров, и оканчивается на EOC). Для директории выделяется кластер (если это не корневая директория FAT16/FAT12), в DIR_FstClusLO и DIR_FstClusHI записывается номер этого кластера, а в таблицу FAT для этого кластера записывается EOC. Затем весь кластер заполняется 0. Если директория является корневой директорией, то на этом всё (в корневой директории нет записей «точка» и «две точки»). Если директория не корневая, то нужно сделать две специальные записи в первых двух 32-байтных структурах (первые две 32 байтные записи, в регионе данных, в кластере который только что создали).

В первой записи директории, поле DIR_Name содержит: "`.`". Во второй записи директории, поле DIR_Name содержит: "`..`". Они называются *точка* и *две точки*. Поле DIR_FileSize в обеих записях содержит 0, а поля даты и времени такие же, как у этой новой директории. Затем для записи *точка* (первая запись) устанавливаются DIR_FstClusLO и DIR_FstClusHI такие же, как у этой новой директории (номер кластера, который содержит эти записи *точка* и *две точки*).

В завершении, для записи *две точки* (это вторая запись) устанавливаются DIR_FstClusLO и DIR_FstClusHI в номер первого кластера родительской директории (значение 0, если она корневая, даже если на FAT32).

Содержание записей *точка* и *две точки*:

- Запись *точка* это директория, указывающая на саму себя.
- Запись *две точки* указывает на первый кластер родительской директории (но 0, если родительская директория является корневой).

Формат Даты и Времени

Многие файловые системы FAT поддерживают Дата/Время только для DIR_WrtTime и DIR_WrtDate. В связи с этим, DIR_CrtTimeMil, DIR_CrtTime, DIR_CrtDate и DIR_LstAccDate практически необязательные поля. Однако DIR_WrtTime и DIR_WrtDate *должны* поддерживаться. Если другие поля даты и времени не поддерживаются, то они должны быть установлены в 0, и игнорироваться при файловых операциях.

Формат Даты. Штамп даты директории FAT – это 16-битное поле, которое относится к эпохе MS-DOS с 01/01/1980. Описание формата (бит 0 младший в 16-битном слове, бит 15 старший в 16-битном слове):

```
Биты 0 – 4: День месяца, допустимый диапазон 1-31 включительно.
Биты 5 – 8: Месяц года, 1 = Январь, допустимый диапазон 1 – 12 включительно.
Биты 9 – 15: Количество лет с 1980, допустимый диапазон 0 – 127 включительно (1980 – 2107).
```

Формат Времени. Штамп времени директории FAT – это 16-битное поле, имеющее кратность 2 секунды. Описание формата (бит 0 младший в 16-битном слове, бит 15 старший в 16-битном слове).

```
Биты 0 – 4: 2-секундный счётчик, допустимый диапазон 0 – 29 включительно (0 – 58 секунд).
Биты 5 – 10: Минуты, допустимый диапазон 0 – 59 включительно.
Биты 11 – 15: Часы, допустимый диапазон 0 – 23 включительно.
```

Допустимый диапазон времени: от полуночи 00:00:00 до 23:59:58.

Fork-FS: модификация FAT32

Структура “wrap” выполнена на основе директории, и поэтому также являйся ни чем иным как “файлом”, состоящим из списка 32-байтных структур. В отличие от директорий, данная структура не содержит “.” и “..” в первых двух записях и имеет иное значение атрибутов.

Когда создаётся “wrap” (файл с установленным атрибутом ATTR_WRAP в поле DIR_Attr), поле DIR_FileSize устанавливается в 0. “Wrap” состоит просто из списка кластеров и оканчивается на EOC. Для “wrap” выделяется кластер, в DIR_FstClusLO и DIR_FstClusHI записывается номер этого кластера, а в таблицу FAT для этого кластера записывается EOC. Затем весь кластер заполняется 0. Оболочке, как и обычным директориям, могут предшествовать записи типа “Long filename”, содержащие имя файла, а также, в случае порождения от измененного файла, количество групп изменений, которые необходимо применить к исходной цепочке файла-родителя.

“Wrap” может быть создан либо в результате привычных файловых операций записи, либо в результате операции fork. В случае использования классических операций, первые несколько записей директории выделяются под счетчик потомков, выполненный на основе структуры типа “Long filename”, а последующая содержит номер первого кластера с данными, во втором случае, первые несколько записей содержат количество требуемых к применению изменений, а последующая запись имеет тип “link” (файл с установленным атрибутом ATTR_LINK в поле DIR_Attr) и содержит номер первого кластера оболочки родительского файла.

Последующие записи используются для хранения изменений, сгруппированных по версиям файла. Каждая группа содержит счетчик потомков и набор файлов, используемых для адресации первых кластеров цепочек изменений (DIR_FstClusLO и DIR_FstClusHI). Если цепочка не оканчивается фактическим концом файла, то данный файл в поле DIR_FileSize содержит адрес следующего кластера для цепочки изменений. Группы цепочек разделяются с помощью указания одинаковых коротких имен файлов внутри группы.

При изменении файла его новая версия создается, если счетчик потомков > 0 или же текущая версия имеет тип “link”. В случае, если на версию файла нет ни одной ссылки, её можно удалить. При манипуляциями с файлами, образованными в результате операции fork, важно следить за правильной установкой значений счетчиков потомков и количества требуемых изменений.

Чтение файла производится с использованием вспомогательной таблицы адресации, полученной путем наложения поверх таблицы FAT сгенерированного по файлам-изменениям массива адресов.

Структура длинных имён FAT

При добавлении длинных имён в FAT, была задача совместимости со старой архитектурой:

- Быть максимально прозрачными для старых версий MS-DOS. Главной целью являлось, чтобы существующие MS-DOS API в старых MS-DOS/Windows, в процессе поиска "find" не находили записи длинных имён. Найти записи длинных имён могут только MS-DOS API функции, использующие FCB с маской любых имён (*.*) и маской любых атрибутов (FFh). В версиях MS-DOS/Windows, идущих после Windows 95, ни какие MS-DOS API не могут даже случайно найти части записей длинных имён.
- Располагаться на диске компактно, рядом с ассоциированным коротким именем. Как мы потом увидим, записи длинных имён непосредственно примыкают к коротким, и их существование не оказывает заметного влияния на производительность системы.
- Обнаруживаясь дисковыми утилитами, не подвергать опасности целостность данных. Дисковые утилиты обычно не используют MS-DOS API для доступа к структурам диска. Обычно они считывают физические или логические сектора с диска, и сами анализируют их содержимое. Эвристическим методом, они могут предпринять шаги по «исправлению» структур, которые они посчитают «повреждёнными». Записи длинных имён были добавлены в систему FAT таким образом, что данные не теряются при «исправлении» их утилитами, ещё не совместимыми с Windows 95 на старых версиях MS-DOS/Windows.

Для того чтобы достигнуть цели оптимального размещения и прозрачности, запись длинного имени определена как короткая, со специальным атрибутом. Как уже было сказано, запись длинной директории это просто запись обычной короткой директории, у которой поле атрибутов имеет такое значение:

```
ATTR_LONG_NAME          ATTR_READ_ONLY |
                        ATTR_HIDDEN |
                        ATTR_SYSTEM |
                        ATTR_VOLUME_ID |
```

Маска для определения записи длинного имени:

```
ATTR_LONG_NAME_MASK     ATTR_READ_ONLY |
                        ATTR_HIDDEN |
                        ATTR_SYSTEM |
                        ATTR_VOLUME_ID |
                        ATTR_DIRECTORY |
                        ATTR_ARCHIVE
```

Когда системе встречается такая запись, она трактуется как часть списка записей директорий, ассоциированных с одной записью короткой директории. Каждая запись длинной директории имеет следующую структуру:

Структура длинной директории FAT

Поле	Сме-щение	Размер (байт)	Описание
LDIR_Ord	0	1	Порядковый номер этой записи в последовательности записей длинных имён, ассоциированных с записью короткого имени в конце списка. Если соответствует маске 0x40 (LAST_LONG_ENTRY), то запись является последней в списке записей длинных имён. Все правильные списки начинаются с данной маской.
LDIR_Name1	1	10	Если LDIR_Type содержит 0, то символы 1-5 длинного имени в данном компоненте. Если LDIR_Type содержит 1 или 2 – хранит младшие 10 байт счетчика.
LDIR_Attr	11	1	Атрибуты – содержит ATTR_LONG_NAME
LDIR_Type	12	1	Если 0, то запись является компонентом длинного имени. 1 – запись является счетчиком потомков. 2 – хранит количество применяемых изменений
LDIR_Chksum	13	1	Контрольная сумма короткого имени (которое располагается в конце списка).
LDIR_Name2	14	12	Если LDIR_Type содержит 0, то символы 6-11 длинного имени в данном компоненте. Если LDIR_Type содержит 1 или 2 – хранит средние 12 байт счетчика.
LDIR_FstClusLO	26	2	Должно быть НОЛЬ. Это ложное значение FAT "первый кластер", и должно быть нулевым для совместимости с дисковыми утилитами. Для длинного имени значение не используется.
LDIR_Name3	28	4	Если LDIR_Type содержит 0, то символы 12-13 длинного имени в данном компоненте. Если LDIR_Type содержит 1 или 2 – хранит старшие 4 байта счетчика.

Организация и ассоциация коротких и длинных имён в директории

Набор записей длинных имён всегда ассоциирован записи короткого имени, которому они предшествуют. Длинные записи существуют в паре с короткими записями, поскольку только короткие записи видны в старых версиях MS-DOS/Windows. Без сопровождающей записи короткого имени, длинные имена были бы совершенно не видны в старых MS-DOS/Windows. Записи длинных имён не могут легально существовать сами по себе. Записи длинного имени, не имеющие соответствующей записи короткого имени, называются «сиротами». Ниже показано, как набор N записей длинного имени ассоциируются с записью короткого имени.

Записи длинного имени всегда непосредственно предшествуют, и физически прилегают к записи короткого имени. Операционная система делает ещё несколько проверок, чтобы убедиться, что длинное имя правильно ассоциировано короткому.

Последовательность записей длинного имени

Запись	**Номер**
N (последняя) запись длинного имени	LAST_LONG_ENTRY (0x40) | N
… дополнительные записи длинного имени	…
1 (первая) запись длинного имени	1
Запись короткого имени, которому предшествует длинное имя	(нет)

Во-первых, каждая составная запись длинного имени последовательно пронумерована, и последняя запись в поле номера имеет специальный флаг – индикатор последней записи списка. Для этого используется поле LDIR_Ord. Первая запись имени содержит в LDIR_Ord значение 1. N (последняя) запись содержит (n OR LAST_LONG_ENTRY). Заметим, что поле LDIR_Ord не может содержать значения 0xE5 и 0x00. Эти значения всегда используются файловой системой для индикации "свободных" и "последней" записи в кластере директории. Для LDIR_Ord не используются значения за пределами этих двух значений. Значения для LDIR_Ord должны быть от 1 до (N or LAST_LONG_ENTRY). Записи с другими значениями файловая система считает "повреждёнными" или "сиротами".

Во-вторых, есть 8-битная контрольная сумма, вычисленная из короткого имени при создании записей длинного и короткого имени. В вычислении используются все 11 символов короткого имени. Контрольная сумма присутствует в каждой записи длинного имени. Если в любой из списка записей длинного имени контрольная сумма не соответствует короткому имени, то длинное имя считается сиротским. Это может случиться, когда диск с длинными именами подключен к старой системе MS-DOS/Windows, в этом случае при переименовании файла/директории переименовывается только короткое имя.

Алгоритм для вычисления контрольной суммы, реализованный на C:

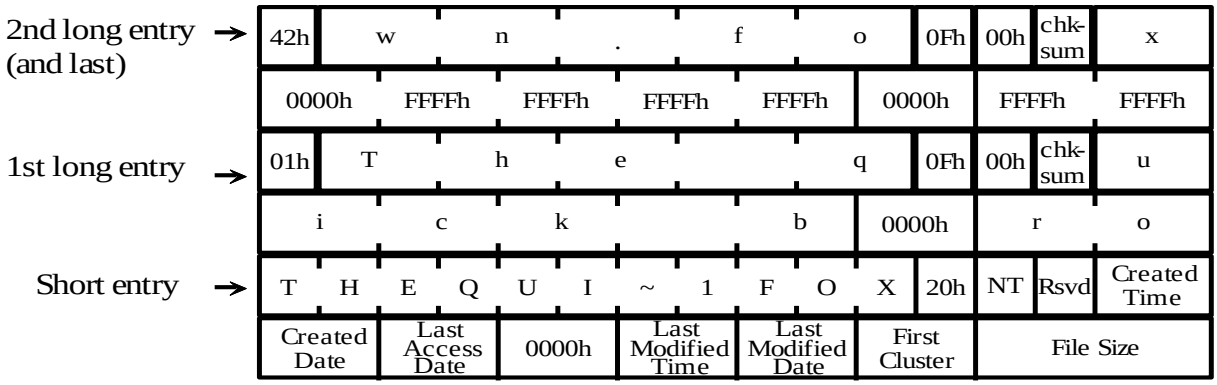
```
//-------------------------------------------------------------------------
// ChkSum()
// Returns an unsigned byte checksum computed on an unsigned byte
// array. The array must be 11 bytes long and is assumed to contain
// a name stored in the format of a MS-DOS directory entry.
// Passed: pFcbName Pointer to an unsigned byte array assumed to be
// 11 bytes long.
// Returns: Sum An 8-bit unsigned checksum of the array pointed
// to by pFcbName.
//-------------------------------------------------------------------------
unsigned char ChkSum (unsigned char *pFcbName)
{
        short FcbNameLen;
        unsigned char Sum;
        Sum = 0;
        for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {
                // NOTE: The operation is an unsigned char rotate right
                Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
        }
        return (Sum);
}
```

Являясь парой, запись короткого имени хранит многие важные поля: дата последнего обращения, время создания, дата создания, номер первого кластера, размер. Также она хранит имя, видимое в старых версиях MS-DOS/Windows. Записи длинного имени могут хранить новую информацию, тут нет необходимости дублировать информацию из записи короткого имени. В основном, записи длинного имени содержат лишь длинное имя файла. Короткое имя, которое ассоциировано длинному имени, называется *вторичным именем*, или просто *псевдоним* файла.

Размещение длинного имени в записях директории

Длинное имя может состоять из большего количества символов, чем умещается в одной записи. В этом случае, имя размещается в нескольких записях. В любом случае, имя разделено на части даже внутри записи. Ниже дан пример, иллюстрирующий, как длинное имя размещается в нескольких записях. После имени ставится NUL, а остальные символы заполняются значением 0xFFFF, специально для обнаружения ошибок от некорректных дисковых утилит. Если в конце имени нет свободных ячеек (например, размер имени кратен 13), то NUL и 0xFFFF не добавляются.

Допустим, файл создан с именем: "`The quick brown.fox`". Данный пример иллюстрирует, как имя упаковано в записях директории. Остальные поля тоже содержат свои значения.



Алгоритм для "авто-генерации" короткого имени из длинного имени, будет дан далее.

Ограничения имён. Разрешённые символы

Короткие имена

Короткие имена ограничены 8 символами до точки (.), и 3 символами расширения. Полный путь с коротким именем не может превышать 80 символов (3 для имени диска + 64 символов пути + 12 для (8.3) имени файла + NUL), включая конечный NUL. Имя может состоять из любой комбинации букв, цифр, и символов с кодом больше 127. Следующие символы тоже разрешены:

```
$ % ' - _ @ ~ ` ! ( ) { } ^ # &
```

Короткие имена хранятся в странице кодировки OEM, действующей в системе на момент создания записи. Записи коротких имён остаются в OEM для совместимости со старыми версиями MS-DOS/Windows. OEM символы это 8-битные символы, и ещё могут быть DBCS пары символов для определённых кодовых страниц.

Короткие имена, передаваемые в файловую систему, всегда преобразуются к верхнему регистру, при этом изначальный регистр теряется. У многих кодировок OEM есть одна проблема – соотношение букв

нижнего и верхнего регистров не 1 к 1. К примеру, несколько букв нижнего регистра отображается на одну букву верхнего регистра. Таким образом, искажается оригинальное имя файла. Также в некоторых случаях это препятствует созданию нормальных файлов с разными именами, поскольку после преобразования в верхний регистр, они становятся одинаковыми.

Длинные имена

Длинные имена ограничены 255 символами, не включая конечный NUL. Полный путь с длинным именем не может превышать 260 символов, включая конечный NUL. Имя может состоять из символов, которые разрешены для коротких имён, плюс символ точки (.) можно использовать многократно. Также разрешён символ «пробел», как и для короткого имени, однако там он обычно не использовался. Следующие шесть символов теперь разрешены в длинных именах (но не в коротких):

```
+ , ; = [ ]
```

Пробелы между символами разрешены. Начальные и конечные пробелы игнорируются.

Начальные и промежуточные точки разрешены, они сохраняются в длинном имени. Конечные точки игнорируются.

Длинные имена хранятся в записях в UNICODE. UNICODE символы – это 16-битные значения. Но нет возможности использовать UNICODE в записях коротких имён, поскольку они рассчитаны на 8-битные символы или DBCS символы.

Длинные имена, передаваемые в файловую систему, не преобразуются к верхнему регистру, при этом изначальный регистр сохраняется. UNICODE может решать проблему многих кодировок OEM, обеспечивая преобразование символов нижнего регистра к верхнему в соотношении 1 к 1.

Сравнение имён, коротких и длинных

Все имена, хранящиеся в записях коротких имён, составляют "пространство коротких имён". Все имена, хранящиеся в записях длинных имён, составляют "пространство длинных имён". Вместе они образуют объединённое пространство имён, где не бывает повторений. Это значит: любое имя в рамках директории, длинное или короткое, встречается только один раз. Хоть в длинных именах и сохраняется оригинальный регистр букв, два имени не могут быть одинаковыми, отличаясь лишь регистром букв. Например, имя "foobar" не может быть создано, если в директории уже есть короткое имя "FOOBAR" или длинное имя "FooBar".

Все операции с использованием поиска в файловой системе (например: find, open, create, delete, rename, fork) регистро-независимы. При открытии файла "FOOBAR" может быть открыт "FooBar" или "foobar", или такой же похожий. Поиском файла по образцу "FOOBAR" будут найдены те же упомянутые файлы. Такие же правила действуют и на буквы дополнительной кодовой страницы.

Операция поиска короткого имени использует только записи коротких имён. Операция поиска длинного имени использует записи длинных и коротких имён. Пока файловая система сканирует директорию, она кэширует составные записи длинного имени. Как только встречается ассоциированная запись короткого имени, операция поиска длинного имени использует для сравнения сначала кэшированное длинное имя, а затем короткое.

Когда символ на диске, сохранённый в кодировке OEM или UNICODE, не может быть транслирован в соответствующий символ кодовой страницы OEM или ANSI, он всегда "транслируется" в "_"

(подчёркивание) при возвращении пользователю – он НЕ модифицируется на диске. Этот символ одинаковый в кодовой странице OEM и ANSI.

Правила именования и Длинные имена

API позволяет вызывающему указать длинное имя для присвоения файлу. API *не* позволяет отдельно задать короткое имя файла. Это запрещено по той причине, что короткие и длинные имена являются объединённым пространством имён. Как мы знаем, пространство имён файловой системы не поддерживает повторяющихся имён. Другими словами, длинное имя файла не может быть таким же, игнорируя регистр, как короткое имя у другого файла. Это ограничение предназначено для предотвращения путаницы у пользователей и программ, касательно правильного имени файла или директории. Чтобы это ограничение сделать прозрачным, при создании нового длинного имени, короткое имя генерируется автоматически из длинного имени, но таким образом, чтобы оно не повторяло других коротких имён.

Данный алгоритм авто-генерации коротких имён из длинных имён разработан после Windows NT.Авто-генерированные короткие имена составляются из *базисного имени* и возможной *конечной цифры*.

Алгоритм генерации базисного имени

Алгоритм генерации базисного имени указан ниже. Этот ***примерный*** алгоритм служит для иллюстрации того, как короткие имена могут авто-генерироваться из длинных имён. Реализация ***должна*** следовать этой базовой последовательности шагов.

```
1. UNICODE имя переданное в файловую систему преобразуется к верхнему регистру.
2. UNICODE имя в верхнем регистре преобразуется в OEM.
Если (в верхнем регистре UNICODE символ отсутствует как символ OEM в кодовой странице OEM)
или (символ OEM не разрешённый для имени 8.3)
{
        Заменить символ на OEM '_' (подчёркивание).
        Установить флаг "неточное преобразование".
}
3. Удалить все начальные и внутренние пробелы из длинного имени.
4. Удалить все начальные точки из длинного имени.
5. Пока (не конец имени) и (символ не точка) и (скопировано символов < 8)
{
        Копировать символ в основную часть базисного имени
}
6. Если после последней точки имени имеется расширение, то в конце основной части базисного
имени добавить точку.
7. Найти последнюю точку в длинном имени.
Если (последняя точка найдена)
{
        Пока (не конец длинного имени) и (скопировано символов < 3)
        {
                Копировать символ в расширенную часть базисного имени
        }
}
8. Приступаем к генерации конечной цифры.
```

Алгоритм генерации конечной цифры

```
Если (не установлен флаг "неточное преобразование") и (длинное имя умещается в формат 8.3)
и (базисное имя не повторяется с существующими короткими именами)
{
        Короткое имя – это только базисное имя, без конечной цифры.
} иначе {
        Добавить конечную цифру "~n" в конец основной части имени, при этом длинна
        основной части сформированного имени не должна превышать 8 символов, а полное имя
        не должно конфликтовать ни с одним существующим коротким именем.
}
```

Строка "~n" может быть в пределах от "~1" до "~999999". Номер "n" выбирается следующим из последовательности файлов с такими же базисными именами. К примеру, имеются следующие короткие имена: LETTER~1.DOC и LETTER~2.DOC. Предполагается, что следующее авто-генерированное имя такого же типа будет LETTER~3.DOC. К примеру, имеются следующие короткие имена: LETTER~1.DOC, LETTER~3.DOC. Теперь следующее авто-генерированное имя такого же типа будет LETTER~2.DOC. Однако, вы *абсолютно не должны* полагаться на это поведение. В директории с очень большим количеством имён с одинаковым типом, алгоритм выбора оптимизируется для скорости, и может выбирать другой "n", базируясь на характеристиках имён которые оканчиваются на "~n", и имеют *такое же* начальное имя.

Влияние записей длинных имён на существующие диски FAT

Поддержка длинных имён наиболее значима для жёстких дисков, но и для гибких дисков тоже есть поддержка. Реализация обеспечивает поддержку длинных имён с сохранением совместимости с существующими дисками FAT. Диск может читаться старыми версиями систем, без каких либо проблем совместимости. Существующий диск не нуждается в изменении формата перед началом использования длинных имён. Все существующие файлы остаются без изменений. Записи длинных имён создаются только при создании длинных имён. Добавление длинного имени к существующему файлу может потребовать перемещения записи 8.3 имени, если требуемое смежное место занято.

В старых системах, записи длинных имён скрыты как «скрытые» или «системные» файлы. Это позволяет защитить неаккуратных пользователей от случайных проблем. Пользователь может копировать файлы, используя 8.3 имя, и записывать новые файлы без каких либо побочных эффектов.

Обратим внимание на то, что происходит с диском FAT при модификации директории в старой системе. Это может сказаться на длинных именах, потому что старые системы игнорируют длинные имена, ассоциированные с 8.3 именами.

Старые системы видят записи длинных имён только при поиске метки диска. При этом, настоящая метка диска будет некорректно определена, если она расположена не первой в корневой директории. Это потому, что у записей длинных имён тоже установлен атрибут «метка диска». Это досадная, но не критичная проблема.

В процессе удаления метки диска, может быть удалена одна из записей длинной директории, хотя это бывает очень редко. Это легко обнаруживается файловой системой. Длинное имя перестанет использоваться от того, что одна или несколько записей помечены удалёнными. Если удалённая запись снова используется, то байт атрибутов уже не будет содержать атрибута длинного имени.

Если файл переименовывается в старой системе, то только короткое имя будет переименовано. Длинное имя останется без изменений. Поскольку в пространстве имён длинные и короткие имена должны быть в согласованном состоянии, то конечно, длинное имя будет неправильным после такого переименования. Контрольная сумма 8.3 имени, которая хранится в длинном имени, служит для выявления таких ошибок. Эта контрольная сумма проверяется перед использованием длинного имени. Переименование может вызвать проблему, только если новое 8.3 имя файла случайно получится с такой же контрольной суммой. Алгоритм контрольной суммы выбран наиболее лучшим для коротких имён.

Переименование 8.3 имени не должно так же конфликтовать с существующими длинными именами. Имеем ввиду, что в старых системах возможно назначить файлу короткое имя, совпадающее с каким-либо длинным именем, без учёта регистра букв. Для предотвращения этого, при автоматическом создании 8.3 имени из длинного имени, которое имеет формат 8.3, это имя напрямую отображается в 8.3 имя путём приведения букв к верхнему регистру.

Если файл удаляется, то длинное имя просто становится сиротским. Если вместо него создаётся новый файл, то длинное имя будет ошибочно ассоциировано новому файлу. Как и в случае переименования, контрольная сумма 8.3 имени предотвращает эту некорректную ассоциацию.

Проверка корректности содержимого директории

Данная информация поможет дисковым утилитам поддерживать 'корректность' каждой записи директории, с учётом возможных будущих нововведений.

1. *НЕ* смотрите в записи директории на поля, помеченные как 'зарезервировано', и если они содержат не ноль, не считайте их 'плохими'.
2. *НЕ* обнуляйте в записи директории поля, помеченные как 'зарезервировано', когда они содержат не ноль (считая их ‘плохими’). Поля являются *зарезервированными,* а не *должны содержать ноль.* Они должны быть проигнорированы вашей программой. Эти поля предназначены для будущих расширений файловой системы. Игнорируя их, утилиты могут нормально работать в будущих версиях операционных систем.
3. *ИСПОЛЬЗУЙТЕ сначала* атрибут A_LONG, когда определяете тип записи директории – длинная или короткая. Ниже дан пример корректного алгоритма:

```
if (((LDIR_attr & ATTR_LONG_NAME_MASK) == ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
    /* Found an active long name sub-component. */
}
```

4. *ИСПОЛЬЗУЙТЕ* биты 4 и 3 для записи короткой директории *вместе,* при определении её типа. Ниже дан пример корректного алгоритма:

```
if (((LDIR_attr & ATTR_LONG_NAME_MASK) != ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
     if((DIR_Attr&(ATTR_DIRECTORY|ATTR_VOLUME_ID|ATTR_WRAP|ARRT_LINK))==0x00)
            /* Found a file. */
     else if((DIR_Attr&(ATTR_DIRECTORY|ATTR_VOLUME_ID|ATTR_WRAP|ARRT_LINK))==ARRT_LINK)
            /* Found a link. */
     else if((DIR_Attr&(ATTR_DIRECTORY|ATTR_VOLUME_ID|ATTR_WRAP|ARRT_LINK))==ATTR_WRAP)
```

```
            /* Found a wrap. */
        else if((DIR_Attr&(ATTR_DIRECTORY|ATTR_VOLUME_ID|ATTR_WRAP|ARRT_LINK))==ATTR_DIRECTORY)
            /* Found a directory. */
        else if((DIR_Attr&(ATTR_DIRECTORY|ATTR_VOLUME_ID|ATTR_WRAP|ARRT_LINK))==ATTR_VOLUME_ID)
            /* Found a volume label. */
        else
            /* Found an invalid directory entry. */
    }
```

5. Используйте значение поля "контрольная сумма" для проверки корректности записи. Поле "первый кластер" в данный момент содержит ноль, но это может быть изменено в будущем.

Несколько заметок относительно директорий FAT

- Записи длинных имён одинаковы у всех типов FAT. Смотрите детальное описание выше.
- Поле DIR_FileSize является 32-битным. Для дисков FAT32, ваш драйвер FAT не должен позволять создавать массив кластеров больше чем 0x100000000 байт, и последний байт последнего кластера для такого размера не должен принадлежать файлу. Таким образом, размер файла не должен быть > 0xFFFFFFFF байт. Это общее ограничение для всех файловых систем FAT. Максимальный разрешённый размер на дисках FAT это 0xFFFFFFFF (4 294 967 295) байт.
- Так же, драйвер FAT не должен допускать директории («контейнер» других файлов) размером более 65 536 * 32 (2 097 152) байт.

ЗАМЕТКА: Это ограничение *не* количества файлов в директории. Это ограничение относится к размеру директории как таковой, и не имеет отношения к содержимому директории. Есть две причины этого ограничения:

1. Поскольку директории FAT не сортированы и не индексированы, то плохая идея создавать огромные директории; в противном случае, такие операции как создание новых записей (которые требуют просмотра каждой существующей записи для проверки наличия такого же имени) станут очень медленными.
2. Есть много драйверов и дисковых утилит FAT, в том числе от Microsoft, которые считывают количество записей директории, используя 16-битную WORD переменную. По этой причине, директории не должны содержать записей больше, чем умещается в 16-битную переменную.

Приложение 2. Результаты тестирования архиваторов. Архивация образа windows_office.vmdk

bzip	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
bzip -1	333	150	1,28	285	99	98	77	99	80
bzip -2	329	153	1,27	287	100	97	79	98	81
bzip -3	349	159	1,27	291	101	96	79	98	83
bzip -4	346	157	1,27	296	103	97	80	97	81
bzip -5	348	156	1,27	302	106	96	79	98	82
bzip -6	356	159	1,27	307	106	96	79	97	80
bzip -7	372	161	1,26	316	105	96	80	97	81
bzip -8	379	164	1,26	325	106	90	80	96	79
bzip -9	376	164	1,26	325	107	96	80	95	81
gz	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
gz -1	104	86	1,34	68	28	82	46	95	91
gz -2	107	79	1,33	69	28	74	44	95	91
gz -3	121	85	1,32	74	27	86	48	96	92
gz -4	120	79	1,3	85	31	87	95	96	94
gz -5	139	81	1,29	91	32	80	52	97	94
gz -6	150	80	1,29	105	31	90	51	96	94
gz -7	175	80	1,29	112	31	90	51	98	94
gz -8	195	82	1,28	147	31	91	51	99	94
gz -9	222	83	1,28	187	31	92	49	99	94
lzop	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
lzo -1	73	86	1,54	20	18	26	27	38	52
lzo -2	85	87	1,54	21	18	26	25	37	53
lzo -3	77	91	1,54	21	18	27	25	38	53
lzo -4	71	90	1,54	23	18	27	27	35	53
lzo -5	85	93	1,54	23	22	21	22	36	44
lzo -6	76	77	1,54	23	22	22	22	35	44
lzo -7	233	65	1,37	161	18	90	27	96	62
lzo -8	390	70	1,37	304	17	95	26	98	61
lzo -9	460	71	1,37	362	17	96	29	98	64
rar	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
rar -1	115	69	1,27			198	38		
rar -2	181	68	1,18			204	42		
rar -3	207	69	1,15			205	42		
rar -4	207	69	1,15			203	41		
rar -5	216	68	1,15			214	43		
zip	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
zip -1	105	87	1,34	58	27	78	41	93	94
zip -2	108	89	1,33	60	27	76	43	93	94
zip -3	117	93	1,32	63	27	87	45	95	94
zip -4	122	91	1,30	73	29	78	47	94	94
zip -5	133	96	1,29	77	29	86	48	96	94
zip -6	149	91	1,29	85	29	87	47	98	94
zip -7	162	95	1,29	95	29	90	47	97	94
zip -8	200	94	1,28	122	29	91	48	98	94
zip -9	254	97	1,28	155	30	93	48	98	94

xz4mb	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
xz4 -1	79	100	1,08	68	62	293	66	338	94
xz4 -2	79	98	1,08	67	62	287	67	339	95
xz4 -3	123	96	1,05	119	59	325	63	338	95
xz4 -4	125	96	1,05	117	59	321	64	341	95
xz4 -5	282	92	0,95	281	55	314	62	341	95
xz4 -6	282	90	0,95	281	55	314	34	316	94
xz4 -7	375	90	0,94	373	54	294	62	316	94
xz4 -8	376	89	0,94	373	54	292	63	296	94
xz4 -9	446	93	0,93	449	54	273	61	295	64
xz64mb	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
xz64 -1	289	91	1,10	278	55	219	61	227	61
xz64 -2	297	92	1,10	286	54	218	60	220	91
xz64 -3	281	92	1,09	272	53	229	60	230	93
xz64 -4	281	91	1,09	271	53	227	62	232	93
xz64 -5	377	89	1,05	680	53	276	63	275	93
xz64 -6	377	90	1,05	683	53	276	62	272	94
xz64 -7	441	88	1,05	450	53	277	63	272	94
xz64 -8	441	89	1,05	446	54	277	62	274	94
xz64 -9	440	89	1,05	455	53	275	62	270	94
xz128mb	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
xz128 -1	1432	98	1,09	1516	59	43	56	40	85
xz128 -2	1755	98	1,09	1415	60	34	56	42	82
xz128 -3	1680	99	1,08	1434	59	35	56	41	85
xz128 -4	1553	99	1,08	1182	59	38	56	50	83
xz128 -5	421	89	1,04	460	53	256	61	227	93
xz128 -6	397	89	1,04	420	54	267	62	252	92
xz128 -7	467	90	1,04	500	53	265	61	249	93
xz128 -8	467	89	1,04	502	53	265	62	247	93
xz128 -9	471	90	1,04	489	52	263	62	255	93
pbzip2	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
pbzip2 -1	111	85	1,28	81	28	341	153	379	372
pbzip2 -2	109	86	1,28	81	28	346	155	380	374
pbzip2 -3	110	84	1,27	84	29	346	164	373	373
pbzip2 -4	124	86	1,27	94	29	313	160	382	375
pbzip2 -5	115	87	1,27	87	30	349	165	378	375
pbzip2 -6	116	87	1,27	90	31	352	171	376	376
pbzip2 -7	110	88	1,27	94	32	352	175	375	376
pbzip2 -8	115	96	1,27	98	34	352	181	377	377
pbzip2 -9	118	97	1,26	100	35	355	185	380	378
pigz	compr time (sec)	decompr time (sec)	compr. size (Gb)	c. time /dev/null (sec)	d. time /dev/null (sec)	cpu c. (%)	cpu d. (%)	cpu c. /dev/null (%)	cpu d. /dev/null (%)
pigz -1	41	71	1,34	25	19	171	38	268	114
pigz -2	40	74	1,33	27	18	179	35	270	114
pigz -3	39	73	1,32	26	18	194	36	288	112
pigz -4	41	72	1,29	30	19	215	42	291	117
pigz -5	42	72	1,29	29	19	230	40	329	118
pigz -6	45	73	1,28	34	19	241	41	318	117
pigz -7	46	70	1,28	36	19	253	42	327	118
pigz -8	56	73	1,28	44	19	274	40	341	118
pigz -9	68	70	1,28	55	19	284	43	350	118

Приложение 3. Результаты тестирования средств дедупликации

	4				8				16				32				64				128				variable						
SDFS	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	dedup ratio	без дедуп (Gb)	с учетом дедуп (Gb)	время (sec)	compress ratio	dedup ratio	занято без учета дедуп (Gb)	свободно из 50Gb без учета дедуп (Gb)	занято с учетом дедуп (Gb)	свободно из 50Gb с учетом дедуп (Gb)	время (sec)
sdfs origin copy	1,3	5	1,5	1230	1,35	5	1,5	660	1,38	5	1,5	300	1,42	5	1,5	210	1,46	5	1,5	150	1,48	5	1,5	150	1	1,29	5	45	0,8	49	780
sdfs office copy	2,38	10	2,5	2160	2,8	10	2,5	810	3,1	10	3	480	3,28	10	3	330	3,44	10	3	240	3,55	10	3	180	1	2,66	10	40	2	48	1320
sdfs office 2nd copy	2,38	15	2,5	210	2,8	15	2,5	180	3,1	15	3	120	3,28	15	3	90	3,44	15	3	60	3,55	15	3	60	1	2,66	15	35	2	48	900
sdfs office recover				510				300				180				120				90				120							540
LESSFS	dedup ratio	без дедуп	с учетом дедуп	время	dedup ratio	без дедуп	с учетом дедуп	время	dedup ratio	без дедуп	с учетом дедуп	время	dedup ratio	без дедуп	с учетом дедуп	время	dedup ratio	без дедуп	с учетом дедуп	время	dedup ratio	без дедуп	с учетом дедуп	время	compress ratio	dedup ratio	без учета дедуп	без учета дедуп	с учетом дедуп	с учетом дедуп	время
lessfs origin copy														5	1,54	540		5	1,52	300		5	1,52	180							
lessfs office copy														10	3,52	780		10	3,56	420		10	3,62	240							
lessfs office 2nd copy														15	3,62	870		15	3,62	420		15	3,65	240							
lessfs office recover																120				90				120							
lessfs qlz origin copy														5	0,95	540		5	0,91	300		5	0,89	180							
lessfs qlz office copy														10	2,3	780		10	2,25	420		10	2,24	240							
lessfs qlz office 2nd copy														15	2,4	840		15	2,3	420		15	2,26	240							
lessfs qlz office recover																150				120				120							
lessfs snappy origin copy														5	0,98	540		5	0,92	300		5	0,89	180							

lessfs snappy office copy														10	2,3	780		10	2,2	420		10	2,25	240							
lessfs snappy office 2nd copy														15	2,4	870		15	2,3	270		15	2,27	240							
lessfs snappy office recover																120				120				120							
lessfs qlz15 origin copy														5	0,95	540		5	0,91	300		5	0,89	180							
lessfs qlz15 office copy														10	2,3	840		10	2,25	420		10	2,24	240							
lessfs qlz15 office 2nd copy														15	2,4	900		15	2,3	420		15	2,27	240							
lessfs qlz15 office recover																150				120				120							
lessfs bzip origin copy														5	0,82	570		5	0,77	330		5	0,7	210							
lessfs bzip office copy														10	2	840		10	1,91	420		10	1,84	270							
lessfs bzip office 2nd copy														15	2,1	900		15	1,96	420		15	1,86	240							
lessfs bzip office recover																240				240				240							
lessfs gzip origin copy														5	0,95	540		5	0,92	300		5	0,85	180							
lessfs gzip office copy														10	2,3	780		10	2,25	420		10	2,2	240							
lessfs gzip office 2nd copy														15	2,4	900		15	2,3	420		15	2,23	240							
lessfs gzip																120				120				120							

office recover																															
lessfs deflate origin copy														5	0,8	540		5	0,77	300		5	0,7	180							
lessfs deflate office copy														10	2	780		10	1,91	420		10	1,85	300							
lessfs deflate office 2nd copy														15	2,1	870		15	1,97	420		15	1,87	300							
lessfs deflate office recover																120				120				120							
ZFS	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**dedup ratio**	**без дед уп**	**с учетом дедуп**	**время**	**compress ratio**	**dedup ratio**	**без учета дед уп**	**без учета дедуп**	**с учетом дедуп**	**с учетом дедуп**	**время**
zfs compression origin copy																									1,56	1	0,95	47,8	0,95	47,8	60
zfs compression office copy																									1,47	1,08	2,62	45,6	2,41	46,3	60
zfs compression office 2nd copy																									1,45	1,78	4,29	45,5	2,42	46,3	60
zfs compression office recover																															60
zfs origin copy																									1	3,32	4,93	46,4	1,49	47,3	90
zfs office copy																									1	2,8	9,95	44,3	3,56	45,2	90
zfs office 2nd copy																									1	4,22	15	44,2	3,56	45,2	60
zfs office recover																															90
zfs lzjb origin copy																									1,56	1	0,95	47	0,95	47,8	90
zfs lzjb office copy																									1,47	1,08	2,63	45,6	2,41	46,3	60

zfs lzjb office 2nd copy																									1,45	1,78	4,29	45,5	2,42	46,3	60
zfs lzjb office recover																															60
zfs gzip origin copy																									2,08	1	0,71	47,3	0,71	48	60
zfs gzip office copy																									1,9	1,09	2,03	46,1	1,86	46,9	90
zfs gzip office 2nd copy																									1,86	1,8	3,33	46,1	1,86	46,9	90
zfs gzip office recover																															60
zfs gzip1 origin copy																									2	1	0,75	47,2	0,75	48	60
zfs gzip1 office copy																									1,83	1,09	2,1	46,1	1,93	46,8	60
zfs gzip1 office 2nd copy																									1,8	1,79	3,46	46	1,94	46,8	60
zfs gzip1 office recover																															60
zfs gzip2 origin copy																									2,02	1	0,74	47,2	0,74	48	60
zfs gzip2 office copy																									1,85	1,09	2,08	46,1	1,91	46,8	60
zfs gzip2 office 2nd copy																									1,81	1,79	3,43	46	1,92	46,8	60
zfs gzip2 office recover																															60
zfs gzip3 origin copy																									2,03	1	0,73	47,3	0,73	48	60
zfs gzip3 office copy																									1,86	1,09	2,07	46,1	1,9	46,8	60
zfs gzip3 office 2nd copy																									1,83	1,79	3,4	46,1	1,9	46,8	60
zfs gzip3 office recover																															60
zfs gzip4 origin copy																									2,06	1	0,72	47,3	0,72	48	60

zfs gzip4 office copy																									1,88	1,09	2,05	46,1	1,88	46,9	60
zfs gzip4 office 2nd copy																									1,84	1,79	3,37	46,1	1,88	46,9	60
zfs gzip4 office recover																															60
zfs gzip5 origin copy																									2,08	1	0,71	47,3	0,71	48	60
zfs gzip5 office copy																									1,9	1,09	2,03	46,1	1,87	46,9	60
zfs gzip5 office 2nd copy																									1,86	1,8	3,34	46,1	1,87	46,9	60
zfs gzip5 office recover																															60
zfs gzip6 origin copy																									2,08	1	0,71	47,3	0,71	48	60
zfs gzip6 office copy																									1,9	1,09	2,02	46,1	1,86	46,9	90
zfs gzip6 office 2nd copy																									1,86	1,8	3,33	46,1	1,86	46,9	60
zfs gzip6 office recover																															60
zfs gzip7 origin copy																									2,09	1	0,71	47,3	0,71	48	60
zfs gzip7 office copy																									1,9	1,09	2,02	46,1	1,86	46,9	90
zfs gzip7 office 2nd copy																									1,87	1,8	3,33	46,1	1,86	46,9	90
zfs gzip7 office recover																															60
zfs gzip8 origin copy																									2,09	1	0,71	47,3	0,71	48	90
zfs gzip8 office copy																									1,91	1,09	2,02	46,1	1,86	46,9	90
zfs gzip8 office 2nd copy																									1,87	1,8	3,33	46,1	1,86	46,9	90
zfs gzip8 office recover																															60

zfs gzip9 origin copy																									2,09	1	0,71	47,3	0,71	48	90
zfs gzip9 office copy																									1,91	1,09	2,02	46,1	1,86	46,9	120
zfs gzip9 office 2nd copy																									1,87	1,8	3,32	46,1	1,86	46,9	90
zfs gzip9 office recover																															60
zfs zle origin copy																									1,14	1	1,31	46,7	1,3	47,4	90
zfs zle office copy																									1,1	1,09	3,49	44,8	3,2	45,6	90
zfs zle office 2nd copy																									1,09	1,77	5,66	44,7	3,2	45	60
zfs zle office recover																															60