Управление вводом-выводом, файловой системой и безопасностью в ОС Windows. Часть 3

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015 15:18:00

История изменений

14.09.2014 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

Лекция 12. Управление вводом-выводом, файловой системой и безопасность в ОС Windows. Часть 3

Вопросы

1. Файловые системы ОС Windows.
2. Архитектура драйвера файловой системы.
3. Анализ проблем в файловой системе.
4. Цели разработки и особенности NTFS.
5. Драйвер файловой системы NTFS.
6. Структура NTFS на диске.
7. Поддержка восстановления в NTFS.
8. Механизм EFS.

Файловые системы ОС Windows

Базовые понятия

Секторы — аппаратно адресуемые блоки носителя. Размер секторов на жестких дисках в x86-системах почти всегда равен 512 байтам. Таким образом, если операционная система должна модифицировать 632-й байт диска, она записывает 512-байтовый блок данных во второй сектор диска.

Форматы файловых систем определяют принципы хранения данных на носителе и влияют на характеристики файловой системы. Например, файловая система, формат которой не допускает сопоставления прав доступа с файлами и каталогами, не поддерживает защиту. Формат файловой системы также может налагать ограничения на размеры файлов и емкости поддерживаемых устройств внешней памяти. Наконец, некоторые форматы файловых систем эффективно реализуют поддержку либо больших, либо малых файлов и дисков.

Кластеры — адресуемые блоки, используемые многими файловыми системами. Размер кластера всегда кратен размеру сектора (Рис. 1). Файловая система использует кластеры для более эффективного управления дисковым пространством: кластеры, размер которых превышает размер сектора, позволяют разбить диск на блоки меньшей длины — управлять такими блоками легче, чем секторами. Потенциальный недостаток кластеров большего размера — менее эффективное использование дискового пространства, или внутренняя фрагментация, которая возникает из-за того, что размеры файлов редко бывают кратны размеру кластера.

Метаданные — это данные, хранящиеся на томе и необходимые для поддержки управления файловой системой. Как правило, они недоступны приложениям. Метаданные включают, например, информацию, определяющую местонахождение файлов и каталогов на томе.

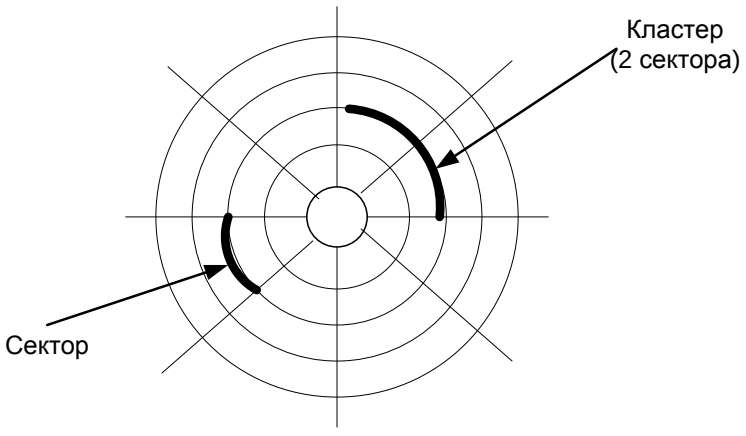


Рис. 1. Секторы и кластер на диске

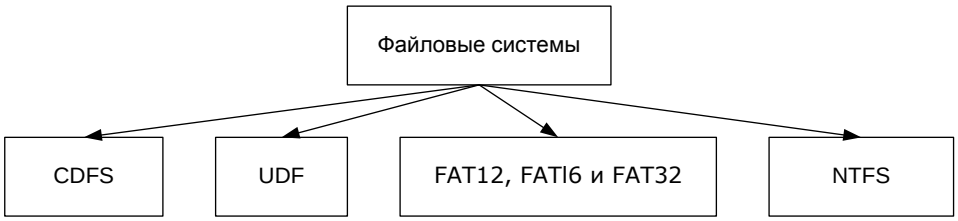


Рис. 2. Файловые системы, которые поддерживает Windows

Каждая из этих файловых систем оптимальна для определенной среды.

CDFS

CDFS, или файловая система CD-ROM (только для чтения), обслуживается драйвером \Windows\System32\Drivers\Cdfs.sys, который поддерживает надмножества форматов ISO-9660 и Joliet. Если формат ISO-9660 сравнительно прост и имеет ряд ограничений, например имена с буквами верхнего регистра в кодировке ASCII и максимальной длиной в 32 символа, то формат Joliet более гибок и поддерживает Unicode-имена произвольной длины. Если на диске присутствуют структуры для обоих форматов (чтобы обеспечить максимальную совместимость), CDFS использует формат Joilet. CDFS присущ ряд ограничений:

- максимальная длина файлов не должна превышать 4 Гб;
- число каталогов не может превышать 65 535.

CDFS считается унаследованным форматом, поскольку индустрия уже приняла в качестве стандарта для носителей, предназначенных только для чтения, формат Universal Disk Format (UDF).

UDF

Файловая система UDF в Windows является UDF-совместимой реализацией OSTA (Optical Storage Technology Association). (UDF является подмножеством формата ISO-13346 с расширениями для поддержки CD-R, DVD-R/RW и т. д.) OSTA определила UDF в 1995 году как формат магнитооптических носителей, главным образом DVD-ROM, предназначенный для замены формата ISO-9660. UDF включен в спецификацию DVD и более гибок, чем CDFS. Драйвер UDF поддерживает UDF версий 1.02 и 1.5 в Windows 2000, а также версий 2.0 и 2.01 в Windows XP и Windows Server 2003. Файловые системы UDF обладают следующими преимуществами:

- длина имен файлов и каталогов может быть до 254 символов в ASCII-кодировке или до 127 символов в Unicode-кодировке;
- файлы могут быть разреженными (sparse);
- размеры файлов задаются 64-битными значениями.

FAT12, FAT16 и FAT32

Windows поддерживает файловую систему FAT по трем причинам:

- для возможности обновления операционной системы с прежних версий Windows до современных,
- для совместимости с другими операционными системами при многовариантной загрузке,
- как формат гибких дисков. Драйвер файловой системы FAT в Windows реализован в \Windows\System32\Drivers\ Fastfat.sys.

В название каждого формата FAT входит число, которое указывает разрядность, применяемую для идентификации кластеров на диске. 12-разрядный идентификатор кластеров в FAT12 ограничивает размер дискового раздела 212 (4096) кластерами. В Windows используются кластеры размером от 512 байтов до 8 Кб, так что размер тома FAT12 ограничен 32 Мб.

FAT16 — за счет 16-разрядных идентификаторов кластеров — может адресовать до 216 (65 536) кластеров. В Windows размер кластера FAT16 варьируется от 512 байтов до 64 Кб, поэтому размер FAT16-тома ограничен 4 Гб. Размер кластеров, используемых Windows, зависит от размера тома (Таблица 1). Если вы форматируете том размером

менее 16 Мб для FAT с помощью команды `format` или оснастки Disk Management (Увправление дисками), Windows вместо FAT16 использует FAT12.

Таблица 1. Размеры кластеров в FAT16 в Windows по умолчанию

Размер тома (Мб)	Размер кластера
0-32	512 байтов
33-64	1 Кб
65-128	2 Кб
129-256	4 Кб
257-511	8 Кб
512-1023	16 Кб
1024-2047	32 Кб
2048-4095	64 Кб

Том FAT делится на несколько областей (Рис. 3). Таблица размещения файлов (file allocation table, FAT), от которой и произошло название файловой системы FAT, имеет по одной записи для каждого кластера тома. Поскольку таблица размещения файлов критична для успешной интерпретации содержимого тома, FAT поддерживает две копии этой таблицы. Так что, если драйвер файловой системы или программа проверки целостности диска (вроде Chkdsk) не сумеет получить доступ к одной из копий FAT (например, из-за плохого сектора на диске), она сможет использовать вторую копию.

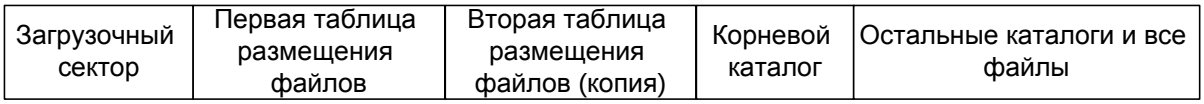


Рис. 3. Организация FAT

Записи в таблице FAT определяют цепочки размещения файлов и каталогов (Рис. 4), где отдельные звенья представляют собой указатели на следующий кластер с данными файла. Элемент каталога для файла хранит начальный кластер файла. Последний элемент цепочки размещения файла содержит зарезервированное значение 0xFFFF для FAT16 и 0xFFF для FAT12. Записи FAT, описывающие свободные кластеры, содержат нулевые значения. На Рис. 4 показан файл FILE1, которому назначены кластеры 2, 3 и 4; FILE2 фрагментирован и использует кластеры 5, 6 и 8; а FILE3 занимает только кластер 7. Чтение файла с FAT-тома может потребовать просмотра больших блоков таблицы размещения файлов для поиска всех его цепочек размещения.

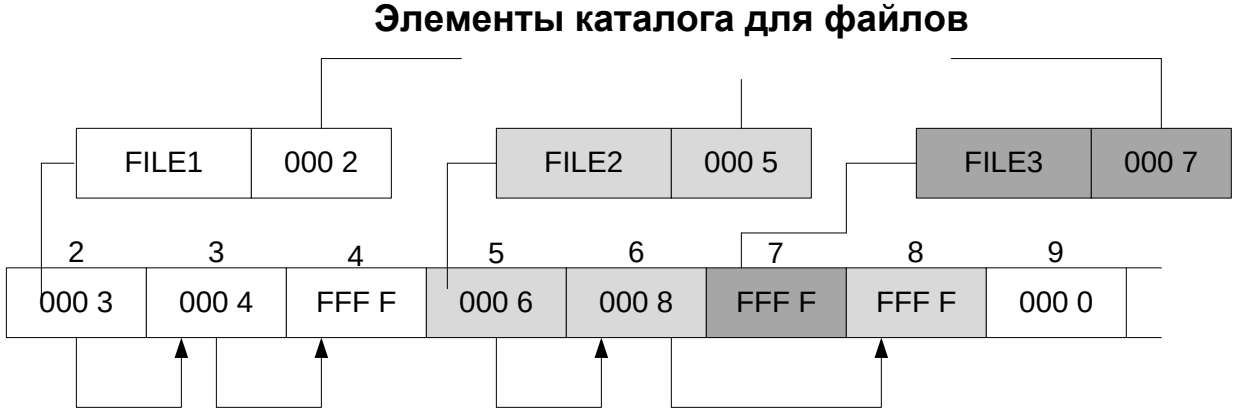


Рис. 4. Пример цепочек размещения файлов в FAT

В начале тома FAT12 или FAT16 заранее выделяется место для корневого каталога, достаточное для хранения 256 записей (элементов), что ограничивает число файлов и каталогов в корневом каталоге (в FAT32 такого ограничения нет). Элемент каталога FAT, размер которого составляет 32 байта, хранит имя файла, его размер, начальный кластер и метку времени (время создания, последнего доступа и т. д.). Если имя файла состоит из Unicode-символов или не соответствует правилам именования по формуле «8.3», принятым в MS-DOS, оно считается длинным и для его хранения выделяются дополнительные элементы каталога. Вспомогательные элементы предшествуют главному элементу для файла. На Рис. 5 показан пример элемента каталога для файла с именем «The quick brown fox». Система создала представление этого имени в формате «8.3», THEQUI~l.FOX (в элементе каталога вы не увидите «.», поскольку предполагается, что точка следует после восьмого символа), и использовала два дополнительных элемента для хранения длинного Unicode-имени. Каждая строка на Рис. 5 состоит из 16 байтов.

Рис. 5. Элементы каталога FAT

FAT32 — более новая файловая система на основе формата FAT; она поддерживается Windows 95 OSR2, Windows 98 и Windows Millennium Edition. FAT32 использует 32-разрядные идентификаторы кластеров, но при этом резервирует старшие 4 бита, так что эффективный размер идентификатора кластера составляет 28 бит. Поскольку максимальный размер кластеров FAT32 равен 32 Кб, теоретически FAT32 может работать с 8-терабайтными томами. Windows ограничивает размер новых томов FAT32 до 32 Гб, хотя поддерживает существующие тома FAT32 большего размера (созданные в других операционных системах). Большее число кластеров, поддерживаемое FAT32, позволяет ей управлять дисками более эффективно, чем FATl6. FAT32 может использовать 512-байтовые кластеры для томов размером до 128 Мб. Размеры кластеров на томах FAT32 по умолчанию показаны в Таблица 2.

Таблица 2. Размер кластеров на томах FAT32 по умолчанию

Размер раздела	Размер кластера
От 32 Мб до 7 Гб	4
8-16 Гб	8
16-32 Гб	16
32 Гб	32

NTFS

NTFS — встроенная («родная») файловая система Windows. NTFS использует 64-разрядные номера кластеров. Это позволяет NTFS адресовать тома размером до 16 экзабайт (16 миллиардов Гб). Однако Windows ограничивает размеры томов NTFS до значений, при которых возможна адресация 32-разрядными кластерами, т. е. до 128 Тб (с использованием кластеров по 64 Кб).

Таблица 3. Размеры кластеров на томах NTFS

Размер тома	Размер кластера
512 Мб и меньше	512 байтов
513-1024 Мб	1 Кб
1025-2048 Мб	2 Кб
Более 2048 Мб	4 Кб

NTFS поддерживает ряд дополнительных возможностей — защиту файлов и каталогов, дисковые квоты, сжатие файлов, символьные ссылки на основе каталогов и шифрование. Одно из важнейших свойств NTFS — восстанавливаемость. При неожиданной остановке системы целостность метаданных тома FAT может быть утрачена, что вызовет повреждение структуры каталогов и значительного объема данных. NTFS ведет журнал изменений метаданных путем протоколирования транзакций, поэтому целостность структур файловой системы может быть восстановлена без потери информации о структуре файлов или каталогов.

Архитектура драйвера файловой системы

Драйвер файловой системы (file system driver, FSD) управляет форматом файловой системы. Хотя FSD выполняются в режиме ядра, у них есть целый ряд особенностей по сравнению со стандартными драйверами режима ядра. Возможно, самой важной особенностью является то, что они должны регистрироваться у диспетчера ввода-вывода и более интенсивно взаимодействовать с ним. Кроме того, для большей производительности FSD обычно полагаются на сервисы диспетчера кэша. Таким образом, FSD используют более широкий набор функций, экспортируемых Ntoskrnl, чем стандартные драйверы. Если для создания стандартных драйверов режима ядра требуется Windows DDK, то для создания драйверов файловых систем понадобится Windows Installable File System (IFS) Kit. В Windows два типа драйверов файловых систем:

- локальные FSD, управляющие дисковыми томами, подключенными непосредственно к компьютеру;
- сетевые FSD, позволяющие обращаться к дисковым томам, подключенным к удаленным компьютерам.

Локальные FSD

К локальным FSD относятся Ntfs.sys, Fastfat.sys, Udfs.sys, Cdfs.sys и Raw FSD (интегрированный в Ntoskrnl.exe). На Рис. 6 показана упрощенная схема взаимодействия локальных FSD с диспетчером ввода-вывода и драйверами устройств внешней памяти. Локальный FSD должен зарегистрироваться у диспетчера ввода-вывода. После регистрации FSD диспетчер ввода-вывода может вызывать его для распознавания томов при первом обращении к ним системы или одного из приложений. Процесс распознавания включает анализ загрузочного сектора тома и, как правило, метаданных файловой системы для проверки ее целостности.

Все поддерживаемые Windows файловые системы резервируют первый сектор тома как загрузочный. Загрузочный сектор содержит достаточно информации, чтобы FSD мог идентифицировать свой формат файловой системы тома и найти любые метаданные, хранящиеся на этом томе.

Распознав том, FSD создает объект «устройство», представляющий смонтированную файловую систему. Диспетчер ввода-вывода связывает объект «устройство» тома, созданный драйвером устройства внешней памяти (далее — объект тома), с объектом «устройство», созданным FSD (далее — объект FSD), через блок параметров тома

(VPB). Это приводит к тому, что диспетчер ввода-вывода перенаправляет через VPB запросы ввода-вывода, адресованные объекту тома, на объект FSD.

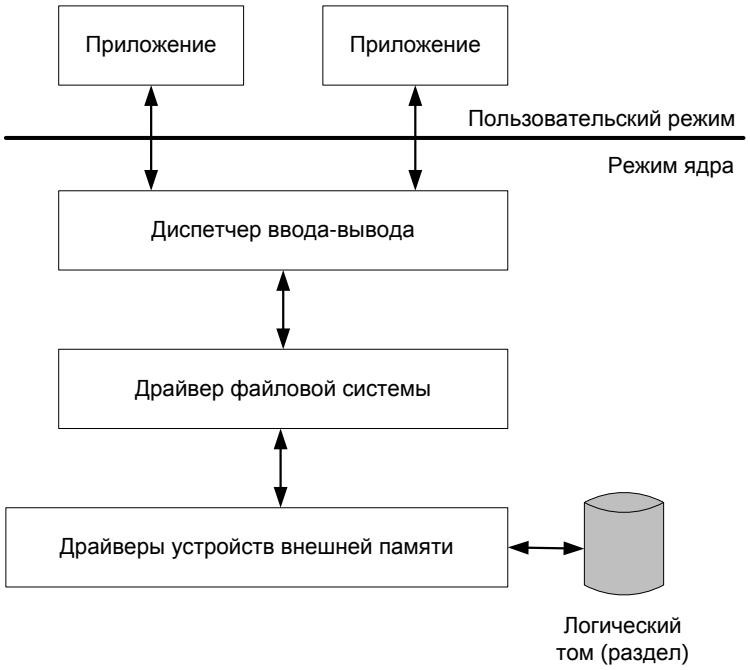


Рис. 6. Локальный FSD

Для большей производительности локальные FSD обычно используют диспетчер кэша, который кэширует данные файловой системы, в том числе ее метаданные. Они также интегрируются с диспетчером памяти, что позволяет корректно реализовать проецирование файлов. Например, всякий раз, когда приложение пытается обрезать файл, они должны запрашивать диспетчер памяти, чтобы убедиться, что за точкой отсечения файл не проецируется ни одним процессом. Windows не разрешает удалять данные файла, проецируемого приложением.

Локальные FSD также поддерживают операции демонтирования файловой системы, позволяющие ОС отсоединять FSD от объекта тома. Демонтирование происходит каждый раз, когда приложение напрямую обращается к содержимому тома или когда происходит смена носителя, сопоставленного с томом. При первом обращении приложения к носителю после демонтирования диспетчер ввода-вывода повторно инициирует операцию монтирования тома для этого носителя.

Удаленные FSD

Удаленные FSD состоят из двух компонентов: клиента и сервера. Удаленный FSD на клиентской стороне позволяет приложениям обращаться к удаленным файлам и каталогам. Клиентский FSD принимает запросы ввода-вывода от приложений и транслирует их в команды протокола сетевой файловой системы, посылаемые через сеть компоненту на серверной стороне, которым обычно является удаленный FSD. Серверный FSD принимает команды, поступающие по сетевому соединению, и выполняет их. При этом он выдает запросы на ввод-вывод локальному FSD, управляющему томом, на котором расположен нужный файл или каталог.

Windows включает клиентский удаленный FSD, LANMan Redirector (редиректор), и серверный удаленный FSD, LANMan Server (сервер) (\Windows\ System32\Drivers\Srv.sys). Редиректор реализован в виде комбинации порт- и минипорт-драйверов, где порт-драйвер (\Windows\System32\Drivers \Rdbss.sys) представляет собой библиотеку подпрограмм, а минипорт-драйвер (\Windows\System32\Drivers\Mrxsmb.sys) использует сервисы, реализуемые порт-драйвером. Еще один минипорт-драйвер редиректора — WebDAV (\Windows\System32\Drivers\Mrxdav.sys), который реализует клиентскую часть поддержки доступа к файлам по HTTR Модель «порт-минипорт» упрощает разработку редиректора, потому что порт-драйвер, совместно используемый всеми минипорт-драйверами удаленных FSD, берет на себя многие рутинные операции, требуемые при взаимодействии между клиентским FSD и диспетчером ввода-вывода Windows.

Взаимодействие между клиентом и сервером при доступе к файлам на серверной стороне через редиректор и серверные FSD показано на Рис. 7.

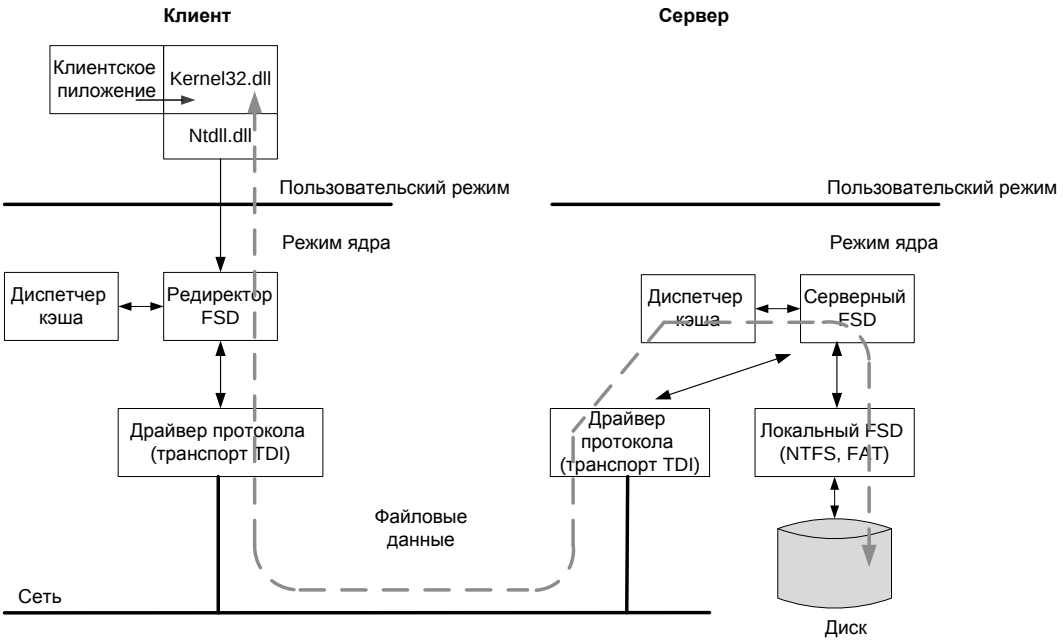


Рис. 7. Обмен файлами по протоколу CIFS

Для форматирования сообщений, которыми обмениваются редиректор и сервер, Windows использует протокол CIFS (Common Internet File System). **CIFS** — это версия протокола Microsoft SMB (Server Message Block).

Удаленные FSD на клиентской стороне обычно используют сервисы диспетчера кэша для локального кэширования файловых данных, относящихся к удаленным файлам и каталогам. Однако удаленный FSD на клиентской стороне должен реализовать протокол поддержки когерентности распределенного кэша, называемый `oplock` (opportunistic locking), гарантирующий, что любое приложение при обращении к удаленному файлу получит те же данные, что и приложения на других компьютерах в сети. Хотя удаленные FSD на серверной стороне участвуют в поддержании когерентности клиентских кэшей, они не кэшируют данные локальных FSD, поскольку те сами кэшируют свои данные.

Когда клиент пытается обратиться к файлу на сервере, он должен сначала запросить `oplock`. Вид доступного клиенту кэширования, определяется типом `oplock`, предоставляемого сервером. Существует три основных типа `oplock`.

- **Level I oplock** предоставляется при монопольном доступе клиента к файлу Клиент, удерживающий для файла oplock этого типа, может кэшировать операции как чтения, так и записи.
- **Level II oplock** — разделяемая блокировка файла. Клиенты, удерживающие oplock этого типа, могут кэшировать операции чтения, но запись в файл делает Level II oplock недействительным.
- **Batch oplock** — самый либеральный тип oplock. Он позволяет клиенту не только читать и записывать файл, но и открывать и закрывать его, не запрашивая дополнительные oplock. Batch oplock, как правило, используется только для поддержки выполнения пакетных (командных) файлов, которые могут неоднократно закрываться и открываться в процессе выполнения.

В отсутствие `oplock` клиент не может осуществлять локальное кэширование ни операций чтения, ни операций записи; вместо этого он должен получать данные с сервера и посылать все изменения непосредственно на сервер.

Проиллюстрировать работу `oplock` поможет пример на Рис. 8. Первому клиенту, открывающему файл, сервер автоматически предоставляет Level I oplock. Редиректор на клиентской стороне кэширует файловые данные при чтении и записи в кэше файловой системы локальной машины. Если тот же файл открывает второй клиент, он также запрашивает Level I oplock. Теперь уже два клиента обращаются к одному и тому же файлу, поэтому сервер должен принять меры для согласования представления данных файла обоим клиентам. Если первый клиент произвел запись в файл (этот

случай и показан на Рис. 8), сервер отзывает oplock и больше не предоставляет его ни одному клиенту. После отзыва `oplock` первый клиент сбрасывает все кэшированные данные файла обратно на сервер.

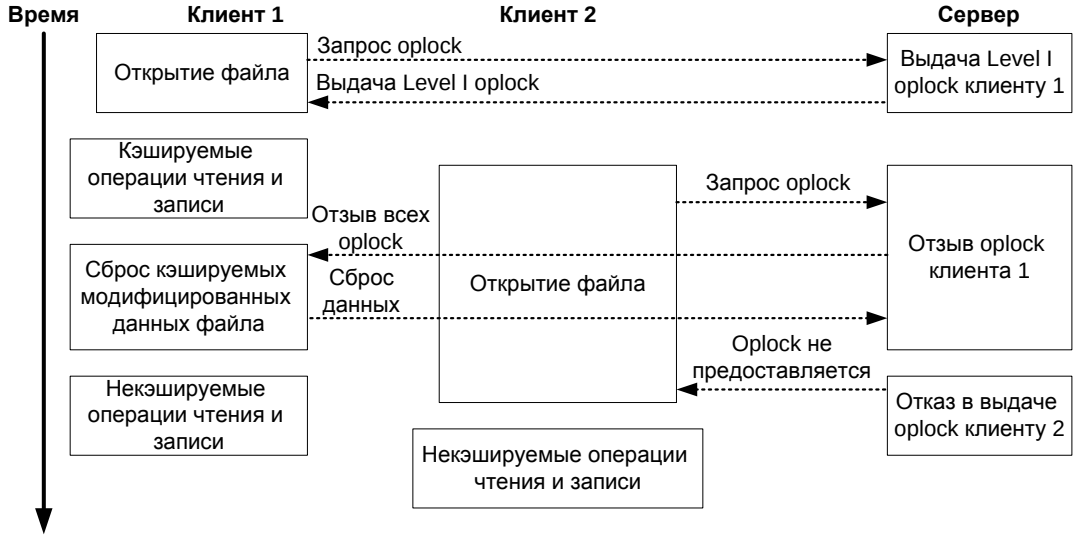


Рис. 8. Пример работы oplock

Если бы первый клиент не произвел запись, его `oplock` был бы понижен до Level II `oplock`, т. е. до `oplock` того же типа, который сервер предоставляет второму клиенту. В этом случае оба клиента могли бы кэшировать операции чтения, но после операции записи любым из клиентов сервер отозвал бы их `oplock`, и последующие операции были бы некэшируемыми. Однажды отозванный, `oplock` больше не предоставляется для этого экземпляра открытого файла. Однако если клиент закрывает файл и повторно открывает его, сервер заново решает, какой `oplock` следует предоставить клиенту. Решение сервера зависит от того, открыт ли файл другими клиентами и производил ли хоть один из них запись в этот файл.

Как работает файловая система

Система и приложения могут обращаться к файлам двумя способами: напрямую (через функции ввода-вывода вроде `ReadFile` и `WriteFile`) и косвенно, путем чтения или записи части своего адресного пространства, где находится раздел проецируемого файла. Упрощенная схема на Рис. 9 иллюстрирует компоненты, участвующие в работе файловой системы, и способы их взаимодействия. Как видите, есть несколько путей вызова FSD:

- из пользовательского или системного потока, выполняющего явную операцию файлового ввода-вывода;
- из подсистем записи модифицированных и спроецированных страниц, принадлежащих диспетчеру памяти;
- неявно из подсистемы отложенной записи, принадлежащей диспетчеру кэша;
- неявно из потока опережающего чтения, принадлежащего диспетчеру кэша;
- из обработчика ошибок страниц, принадлежащего диспетчеру памяти.

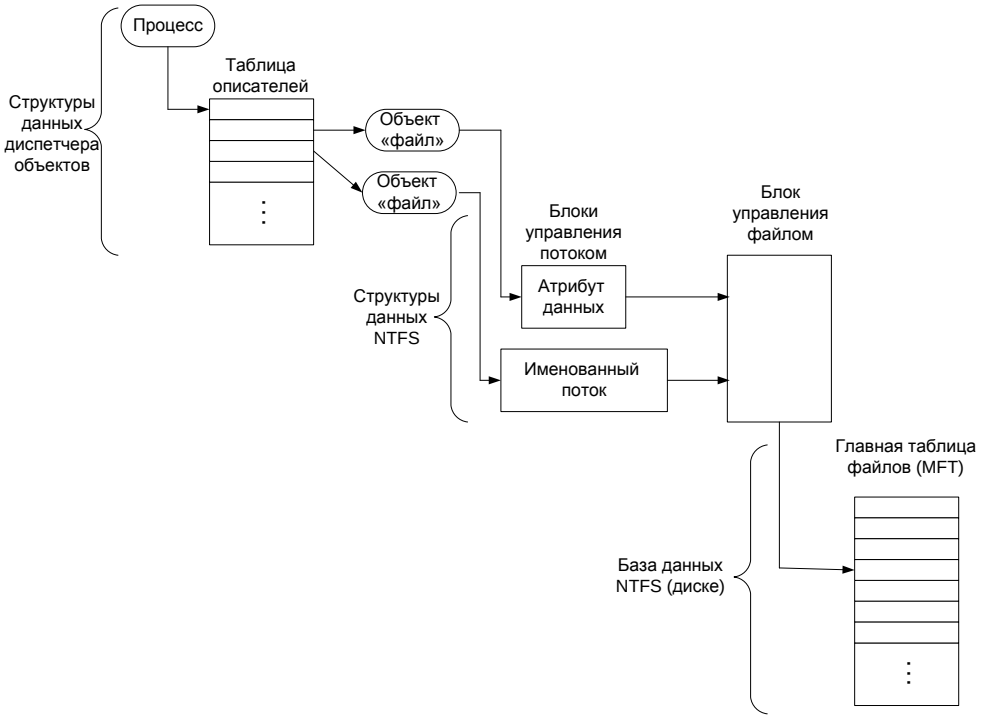


Рис. 9. Компоненты, участвующие в операциях ввода-вывода файловой системы

Явный файловый ввод-вывод

Наиболее очевидный способ доступа приложения к файлам — вызов Windows-функций ввода-вывода, например `CreateFile`, `ReadFile` и `WriteFile`. Приложение открывает файл с помощью `CreateFile`, а затем читает, записывает и удаляет его, передавая описатель файла, возвращенный `CreateFile`, другим Windows-функциям. `CreateFile`, реализованная в Kernel32.dll, вызывает встроенную функцию `NtCreateFile` и формирует полное имя файла, обрабатывая символы «.» и «…» и предваряя путь строкой «\??» (например, \??\C: \Daryl\Todo.txt).

Чтобы открыть файл, системный сервис `NtCreateFile` вызывает функцию `ObOpenObjectByName`, которая выполняет разбор имени, начиная с корневого каталога диспетчера объектов и первого компонента полного имени («??»).

Первое, что делает диспетчер объектов, — транслирует \?? в каталог пространства имен, индивидуальный для сеанса, в котором выполняется данный процесс; на этот каталог ссылается поле DosDevicesDirectory в структуре карты устройств (device map structure) в объекте «процесс». В системах Windows 2000 без Terminal Services поле DosDevicesDirectory ссылается на каталог \?? а в системах Windows 2000 без Terminal Services карта устройств ссылается на индивидуальный для каждого сеанса каталог, где хранятся объекты «символьная ссылка», представляющие все действительные буквы дисков для томов. Однако в Windows XP и Windows Server 2003 в таком каталоге обычно содержатся лишь имена томов для общих сетевых ресурсов, поэтому в этих ОС диспетчер объектов, не найдя имя (в данном примере — G) в индивидуальном для сеанса каталоге, переходит к поиску в каталоге, на который ссылается поле GlobalDosDevicesDirectory карты устройств, сопоставленной с индивидуальным для сеанса каталогом. GlobalDosDevicesDirectory всегда указывает на каталог \Global?? где Windows XP и Windows Server 2003 хранят буквы дисков для локальных томов. Символьная ссылка для буквы диска, присвоенной тому, указывает на объект тома в каталоге \Device, поэтому диспетчер объектов, распознав объект тома, передает остаток строки с именем в функцию `IopParseDevice`, зарегистрированную диспетчером ввода-вывода для объектов «устройство». На Рис. 10 показано, как происходит доступ к объектам томов через пространство имен диспетчера объектов. В данном случае символьная ссылка \??\C: указывает на объект тома \Device\HarddiskVolumel.

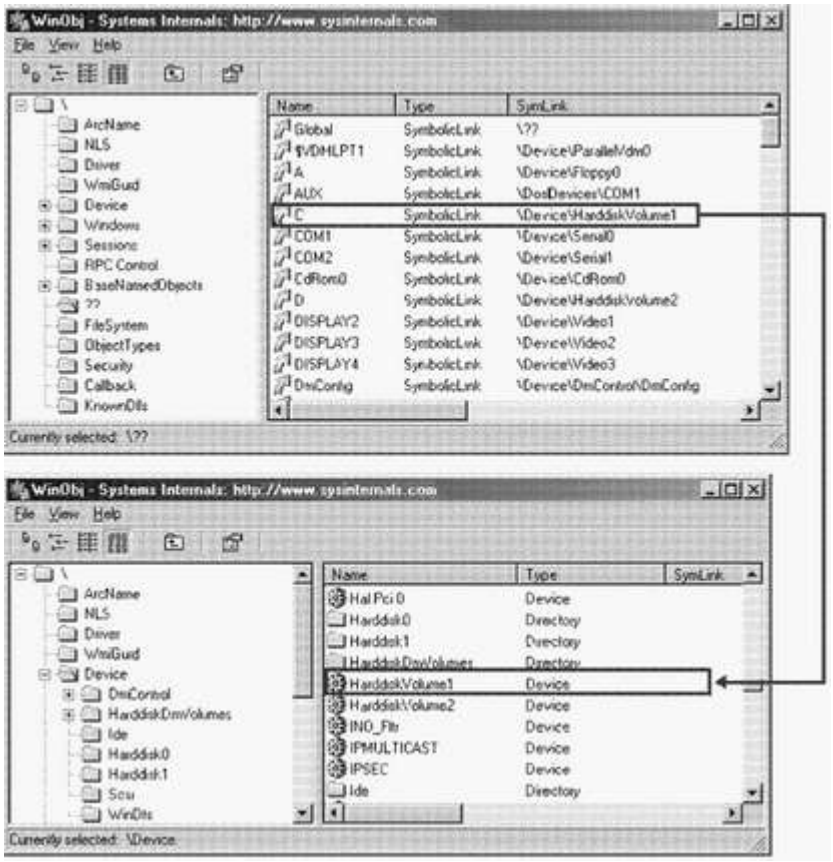


Рис. 10. Разрешение букв дисков

Заблокировав контекст защиты вызывающего потока и получив информацию о защите из его маркера, `IopParseDevice` генерирует пакет запроса ввода-вывода (IRP) типа IRP_MJ_CREATE, создает объект «файл», в котором запоминается имя открываемого файла, и по ссылке в VPB объекта тома находит объект «устройство» смонтированной файловой системы тома. Далее, используя `IoCallDriver`, она передает IRP драйверу файловой системы, которому принадлежит данный объект «устройство».

Когда FSD получает IRP типа IRP_MJ_CREATE, он ищет указанный файл, проверяет права доступа и, если файл есть и пользователь имеет права на запрошенный вид доступа к файлу, возвращает код успешного завершения. Диспетчер объектов создает в таблице описателей, принадлежащей процессу, описатель объекта «файл», который передается назад по цепочке вызовов, в конечном счете достигая приложения в виде параметра, возвращаемого `CreateFile`. Если файловой системе не удается создать файл, диспетчер ввода-вывода удаляет созданный для него объект «файл».

В выполнении `ReadFile` ядро участвует в той же мере, что и в выполнении `CreateFile`, но в этом случае системному сервису `NtReadFiIe` не приходится искать имя — он вызывает диспетчер объектов для трансляции описателя, переданного `ReadFile`, в указатель на объект «файл». Если описатель открытого файла указывает на наличие у вызывающего потока прав на чтение файла, `NtReadFile` создает IRP типа IRPMJREAD и посылает его драйверу файловой системы, в которой находится файл. `NtReadFile` получает объект FSD, хранящийся в объекте «файл», и вызывает `IoCallDriver`. Диспетчер ввода-вывода находит FSD с помощью объекта FSD и передает IRP этому драйверу файловой системы.

Если считываемый файл может быть кэширован, FSD проверяет, инициировано ли кэширование для этого объекта «файл». Если да, поле PrivateCacheMap объекта «файл» указывает на структуру закрытой карты кэша. В ином случае поле PrivateCacheMap будет пустым. Кэширование объекта «файл» инициируется FSD при первой операции записи или чтения над этим объектом, для чего FSD вызывает функцию `CcInitializeCacheMap` диспетчера кэша, и диспетчер кэша создает закрытую и общую карты кэша, а также объект «раздел» (если это еще не сделано).

Убедившись, что кэширование файла разрешено, FSD копирует данные запрошенного файла из виртуальной памяти диспетчера кэша в буфер, указатель на который передан функции `ReadFile` вызывающим потоком. Файловая система выполняет копирование в

рамках блока try/except, что позволяет перехватывать все ошибки, которые могут возникнуть, если приложение указало неверный буфер. Для копирования файловая система использует функцию `CcCopyRead` диспетчера кэша, которая принимает в качестве параметров объект «файл», смещение внутри файла и длину данных.

Диспетчер кэша, выполняя CcCopyRead, получает указатель на общую карту кэша, хранящуюся в объекте «файл». Эта карта хранит указатели на блоки управления виртуальными адресами (VACB) и один элемент VACB соответствует 256-килобайтному блоку файла. Если VACB-указатель для считываемой части файла пуст, CcCopyRead создает VACB, резервируя в виртуальном адресном пространстве диспетчера кэша 256-ки-лобайтное представление, и проецирует на это представление указанную порцию файла (с помощью MmMapViewInSystemCacbe). Затем CcCopyRead просто копирует данные файла из спроецированного представления в переданный ей буфер (буфер, изначально переданный в ReadFile). Если файловых данных в физической памяти нет, операция копирования вызывает ошибки страниц, обслуживаемые MmAccessFault.

Когда возникает ошибка страницы, MmAccessFault изучает виртуальный адрес, вызвавший ошибку, и находит дескриптор виртуального адреса (VAD) в дереве VAD вызвавшего ошибку процесса. В данном случае VAD описывает представление считываемого файла, проецируемое диспетчером кэша, поэтому для обработки ошибки страницы, вызванной действительным виртуальным адресом, MmAccessFault вызывает MiDispatcbFault, которая сначала находит область управления (на нее указывает VAD) и уже через нее отыскивает объект «файл», представляющий открытый файл. (Если файл открывался более чем один раз, возможно наличие списка объектов «файл», связанных указателями в закрытых картах кэша.)

Найдя объект «файл», MiDispatcbFault вызывает функцию IoPageRead диспетчера ввода-вывода, чтобы создать IRP (типа IRP_MJ_READ), и посылает этот IRP к FSD, владеющему объектом «устройство», на который указывает объект «файл». Таким образом, осуществляется повторный вход в файловую систему для чтения данных, запрошенных через CcCopyRead, но на этот раз в IRP присутствует флаг, который сообщает о необходимости некэшируемого и связанного с подкачкой ввода-вывода. Этот флаг сигнализирует FSD, что он должен извлечь данные непосредственно с диска, и тот так и поступает, определяя, какие кластеры диска содержат запрошенные данные, и посылая соответствующие IRP диспетчеру томов, владеющему объектом тома, на котором находится файл. Поле блока параметров тома (VPB) объекта FSD указывает на объект тома.

Диспетчер виртуальной памяти ждет, когда FSD завершит чтение, а потом возвращает управление диспетчеру кэша, который продолжает операцию копирования, прерванную ошибкой страницы. По окончании работы `CcCopyRead` драйвер файловой системы возвращает управление потоку, вызвавшему `NtReadFile`; на этот момент данные из запрошенного файла уже скопированы в буфер потока.

`WriteFile` работает аналогичным образом с тем исключением, что системный сервис NtWriteFile генерирует IRP типа IRP_MJ_WRITE, а FSD вызывает не `CcCopyRead`, а `CcCopyWrite`. Последняя, как и `CcCopyRead`, проверяет, спроецированы ли на кэш части записываемого файла, и копирует в кэш содержимое буфера, переданного в `WriteFile`.

Если файловые данные уже хранятся в системном рабочем наборе, вышеописанный сценарий немного меняется. Если файловые данные находятся в кэше, `CcCopyRead` не вызывает ошибки страниц.

Драйверы фильтров файловой системы

Драйвер фильтра, занимающий в иерархии более высокий уровень, чем драйвер файловой системы, называется драйвером фильтра файловой системы (file system filter driver). Его способность видеть все запросы к файловой системе и при необходимости модифицировать или выполнять их, делает возможным создание таких приложений, как службы репликации удаленных файлов, шифрования файлов, резервного копирования и лицензирования. В любой коммерческий антивирусный сканер, проверяющий файлы на «лету», входит драйвер файловой системы, который перехватывает IRP-пакеты с командами IRP_MJ_CREATE, выдаваемыми при каждом открытии файла приложением. Прежде чем передать такой IRP драйверу файловой системы, которому адресована данная команда, антивирусный сканер проверяет открываемый файл на наличие вирусов. Если файл чист, антивирусный сканер передает IRP дальше по цепочке, но если файл заражен, сканер обращается к своему сервисному процессу для удаления или лечения этого файла. Если вылечить файл

нельзя, драйвер фильтра отклоняет IRP (обычно с ошибкой «доступ запрещен»), чтобы вирус не смог активизироваться.

Анализ проблем в файловой системе

Если в реестре появляются какие-то проблемы, скажем, из-за неправильно сконфигурированной защиты или недостающих параметров/разделов реестра, то они могут быть причиной многих сбоев в работе системы и приложений. Помимо этого, система и приложения используют файлы для хранения данных и обращаются к образам DLL и исполняемых файлов. Значит, ошибки в конфигурации защиты NTFS и отсутствие каких-либо файлов или каталогов также являются распространенной причиной сбоев в работе системы и приложений. А все потому, что система и приложения часто полагаются на возможность беспрепятственного доступа к таким файлам и начинают вести себя непредсказуемым образом, если эти файлы оказываются недоступны.

Утилита Filemon отражает все файловые операции по мере их выполнения, что превращает ее в идеальный инструмент для анализа сбоев системы и приложений из-за проблем в файловой системе. Пользовательский интерфейс Filemon практически идентичен таковому в Regmon, и Filemon включает те же средства фильтрации, выделения и поиска, что и Regmon. Первый запуск Filemon в системе требует учетной записи с теми же привилегиями, что и в случае Regmon: Load Driver и Debug. После загрузки драйвер остается резидентным в памяти, поэтому для последующих запусков Filemon достаточно привилегии Debug.

Базовый и расширенный режимы Filemon

При запуске Filemon начинает работу в базовом режиме, в котором отображаются те операции в файловой системе, которые наиболее полезны для анализа проблем. В этом режиме Filemon не показывает определенные операции в файловой системе, в том числе:

- обращения к файлам метаданных NTFS;
- операции в процессе System;
- ввод-вывод, связанный со страничным файлом;
- ввод-вывод, генерируемый процессом Filemon;
- неудачные попытки быстрого ввода-вывода.

Кроме того, в базовом режиме Filemon сообщает об операциях файлового ввода-вывода, используя описательные имена, а не типы IRP, которые на самом деле представляют их. В частности, операции IRP_MJ_WRITE и FASTIO_ WRITE отображаются как Write, а операции IRP_MJ_CREATE — как Open (при открытии существующих файлов) или Create (при создании новых файлов).

Методики анализа проблем с применением Filemon

Два основных метода анализа проблем с применением Filemon идентичны таковым при использовании Regmon: поиск последней операции в трассировочной информации Filemon перед тем, как в приложении произошел сбой, или сравнение трассировочной информации Filemon для сбойного приложения с аналогичными сведениями для работающей системы.

Обращайте внимание на записи в выводе Filemon со значениями FILE NOT FOUND, NO SUCH FILE, PATH NOT FOUND, SHARING VIOLATION и ACCESS DENIED в столбце Result. Первые три значения указывают на то, что приложение или система пытается открыть несуществующий файл или каталог. Во многих случаях эти ошибки не свидетельствуют о серьезной проблеме. Например, если вы запускаете какую-то программу из диалогового окна Run (Запуск программы), не задавая полный путь к ней, Explorer будет искать эту программу в каталогах, перечисленных в переменной окружения PATH, пока не найдет нужный образ или не закончит просмотр всех перечисленных каталогов. Каждая попытка найти образ в каталоге, где такого образа нет, отражается в выводе Filemon строкой, аналогичной приведенной ниже:

```
5:28:26       PUEXPLORER.EXE:       1568FASTIO_QUERY_OPENC:\Documents       and
Settings\mark.AUSTIN\Start Menu\test.exe  FILE NOT FOUND Attributes: Error
```

Ошибки, связанные с отклонением попыток доступа, — частая причина сбоев приложений при работе с файловой системой, и они возникают, когда у приложения нет соответствующего разрешения на открытие файла или каталога. Некоторые

приложения не проверяют коды ошибок или не обрабатывают ошибки, из-за чего происходит их крах или аварийное завершение, а некоторые выводят при этом сообщения о других ошибках, которые лишь маскируют истинную причину неудачи файловой операции.

Прорехи в защите из-за переполнения буфера представляют серьезную угрозу безопасности, но код результата BUFFER OVERFLOW — просто способ, используемый драйвером файловой системы, чтобы сообщить приложению о нехватке места в выделенном буфере для сохранения полученных данных. Разработчики приложений применяют этот способ, чтобы определить правильный размер буфера, так как драйвер файловой системы заодно сообщает и эту информацию. За операциями с кодом результата BUFFER OVERFLOW обычно следуют операции с успешным результатом.

Примеры выявления истинной причины ошибки с помощью Filemon подробно описано в [3].

Цели разработки и особенности NTFS

Требования к файловой системе класса «high end»

С самого начала разработка NTFS велась с учетом требований, предъявляемых к файловой системе корпоративного класса. Чтобы свести к минимуму потери данных в случае неожиданного выхода системы из строя или ее краха, файловая система должна гарантировать целостность своих метаданных. Для защиты конфиденциальных данных от несанкционированного доступа файловая система должна быть построена на интегрированной модели защиты. Наконец, она должна поддерживать защиту пользовательских данных за счет программной избыточности данных в качестве недорогой альтернативы аппаратным решениям. Здесь вы узнаете, как эти возможности реализованы в NTFS.

Восстанавливаемость

В соответствии с требованиями к надежности хранения данных и доступа к ним NTFS обеспечивает восстановление файловой системы на основе концепции атомарной транзакции (atomic transaction). **Атомарные транзакции** — это метод обработки изменений в базе данных, при котором сбои в работе системы не нарушают корректности или целостности базы данных. Суть атомарных транзакций заключается в том, что некоторые операции над базой данных, называемые **транзакциями**, выполняются по принципу «все или ничего». Отдельные изменения на диске, составляющие транзакцию, выполняются **атомарно**: в ходе транзакции на диск должны быть внесены все требуемые изменения. Если транзакция прервана аварией системы, часть изменений, уже внесенных к этому моменту, нужно отменить. Такая отмена называется **откатом** (roll back). После отката база данных возвращается в исходное согласованное состояние, в котором она была до начала транзакции.

NTFS использует атомарные транзакции для реализации возможности восстановления файловой системы. Если некая программа инициирует операцию ввода-вывода, которая изменяет структуру NTFS-тома, т. е. модифицирует структуру каталогов, увеличивает длину файла, выделяет место под новый файл и др., то NTFS обрабатывает такую операцию как атомарную транзакцию. NTFS гарантирует, что транзакция будет либо полностью выполнена, либо отменена, если хотя бы одну из операций не удастся завершить из-за сбоя системы.

Защита

Защита в NTFS построена на модели объектов Windows. Файлы и каталоги защищены от доступа пользователей, не имеющих соответствующих прав. Открытый файл реализуется в виде объекта «файл» с дескриптором защиты, хранящимся на диске как часть файла. Прежде чем процесс сможет открыть описатель какого-либо объекта, в том числе объекта «файл», система защиты Windows должна убедиться, что у этого процесса есть соответствующие полномочия. Дескриптор защиты в сочетании с требованием регистрации пользователя при входе в систему гарантирует, что ни один процесс не получит доступа к файлу без разрешения системного администратора или владельца файла.

Избыточность данных и отказоустойчивость

Восстанавливаемость NTFS действительно гарантирует, что файловая система тома останется доступной, но не дает гарантии полного восстановления пользовательских файлов. Последнее возможно за счет поддержки избыточности данных.

Избыточность данных для пользовательских файлов реализуется через многоуровневую модель драйверов Windows, которая поддерживает отказоустойчивые диски. При записи данных на диск NTFS взаимодействует с диспетчером томов, а тот — с драйвером жесткого диска. Диспетчер томов может зеркалировать, или дублировать, данные одного диска на другом и таким образом позволяет при необходимости использовать данные с избыточной копии. Поддержка таких функций обычно называется RAID уровня 1. Диспетчеры томов также могут записывать данные в чередующиеся области (stripes) на три и более дисков, используя один диск для хранения информации о четности. Если данные на одном диске потеряны или стали недоступными, драйвер может реконструировать содержимое диска с помощью логической операции `XOR`. Такая поддержка называется RAID уровня 5.

Дополнительные возможности NTFS

NTFS — это не только восстанавливаемая, защищенная, надежная и эффективная файловая система, способная работать в системах повышенной ответственности. Она поддерживает ряд дополнительных возможностей (некоторые из них доступны приложениям через API-функции, другие являются внутренними):

- множественные потоки данных;
- имена на основе Unicode;
- универсальный механизм индексации;
- динамическое переназначение плохих кластеров;
- жесткие связи и точки соединения;
- сжатие и разреженные файлы;
- протоколирование изменений;
- квоты томов, индивидуальные для каждого пользователя;
- отслеживание ссылок;
- шифрование;
- поддержка POSIX;
- дефрагментация;
- поддержка доступа только для чтения.

Множественные потоки данных

В любом файле NTFS по умолчанию имеется один безымянный поток данных. Приложения могут создавать дополнительные, именованные потоки данных и обращаться к ним по именам. Чтобы не изменять Windows-функции API ввода-вывода, которым имя файла передается как строковый аргумент, имена потоков данных задаются через двоеточие (:) после имени файла, например:

```
myfile.dat: stream2
```

Каждый поток имеет свой **выделенный размер** (объем зарезервированного для него дискового пространства), **реальный размер** (использованное число байтов) и **длину действительных данных** (инициализированная часть потока). Кроме того, каждому потоку предоставляется отдельная файловая блокировка, позволяющая блокировать диапазоны байтов и поддерживать параллельный доступ.

Множественные потоки данных использует, например, компонент поддержки файлового сервера Apple Macintosh, поставляемый с Windows Server. Системы Macintosh используют два потока данных в каждом файле: один — для хранения данных, другой — для хранения информации о ресурсах. Поскольку NTFS поддерживает множественные потоки данных, пользователь Macintosh может скопировать целую папку Macintosh на сервер Windows, а другой пользователь Macintosh может скопировать ее с сервера без потери информации о ресурсах.

Windows Explorer — еще одно приложение, использующее такие потоки. Когда вы щелкаете правой кнопкой мыши файл NTFS и выбираете команду Properties, на экране появляется диалоговое окно, вкладка Summary (Сводка) которого позволяет сопоставить с файлом такую информацию, как заголовок, тема, имя автора и ключевые слова. Windows Explorer хранит эту информацию в альтернативном потоке под названием Summary Information, добавляемом к файлу.

Имена на основе Unicode

NTFS полностью поддерживает Unicode, используя Unicode-символы для хранения имен файлов, каталогов и томов. Unicode, 16-битная кодировка символов, обеспечивает уникальное представление любого символа основных языков мира, что упрощает обмен информацией между странами. Поскольку в Unicode имеется уникальное представление каждого символа, последний не зависит от того, какая кодовая страница загружена в ОС. Длина имени каждого каталога или файла в пути может достигать 255 символов; в нем могут быть символы Unicode, пробелы и несколько точек.

Универсальный механизм индексации

Архитектура NTFS позволяет индексировать атрибуты файлов на дисковом томе. Это дает возможность файловой системе вести эффективный поиск файлов по неким критериям, например находить все файлы в определенном каталоге. Файловая система FAT индексирует имена файлов, но не сортирует их, что замедляет просмотр больших каталогов.

Некоторые функции NTFS используют преимущества универсальной индексации, в том числе консолидированных дескрипторов защиты, где дескрипторы защиты файлов и каталогов на томе хранятся в едином внутреннем потоке без дубликатов; при этом они индексируются с использованием внутреннего идентификатора защиты, определяемого NTFS.

Динамическое переназначение плохих кластеров

Если диск отформатирован как отказоустойчивый том NTFS, специальный драйвер Windows динамически считывает «хорошую» копию данных, хранившихся в плохом секторе, и посылает NTFS предупреждение о плохом секторе. NTFS выделяет новый кластер, заменяющий тот, в котором находится плохой сектор, и копирует данные в этот кластер. Плохой кластер помечается как аварийный и больше не используется. Восстановление данных и динамическое переназначение плохих кластеров особенно полезно для файловых серверов и отказоустойчивых систем, а также для всех приложений, в которых потеря данных недопустима. Если на момент появления плохого сектора диспетчер томов не был загружен, NTFS все равно заменяет кластер и не допускает его повторного использования, хотя восстановить данные из плохого сектора уже не удастся.

Жесткие связи и точки соединения

Жесткие связи (hard links) позволяют ссылаться на один и тот же файл по нескольким путям (для каталогов жесткие связи не поддерживаются). Процессы могут создавать жесткие связи вызовом Windows-функции `CreateHardLink` или POSIX-функции `ln`.

В дополнение к жестким связям NTFS поддерживает другой тип перенаправления — точки соединения (junctions), также называемые **символьными ссылками** (symbolic links). Они позволяют перенаправлять трансляцию имени файла или каталога из одного каталога в другой. Например, если путь C: \Drivers — ссылка, которая перенаправляет в C: \Windows\System32\Drivers, то приложение, читающее C: \Drivers\Ntfs.sys, на самом деле читает C: \Windows\System\Drivers\Ntfs.sys. Точки соединения представляют собой удобное средство «подъема» каталогов, расположенных слишком глубоко в дереве каталогов, на более высокий уровень, не нарушая исходной структуры или содержимого дерева каталогов. Так, в предыдущем примере каталог драйверов «поднят» на два уровня по сравнению с реальным. Точки соединения неприменимы к удаленным каталогам — они используются только для каталогов на локальных томах.

Точки соединения опираются на механизм NTFS «точки повторного разбора». **Точка повторного разбора** (reparse point) — это файл или каталог, с которым сопоставлен блок данных, называемых **данными повторного разбора** (reparse data); они представляют собой пользовательские данные о файле или каталоге, например о его состоянии или местонахождении. Эти данные могут быть считаны из точки повторного разбора приложением, которое создало их, драйвером файловой системы или диспетчером ввода-вывода. Обнаружив точку повторного разбора при поиске файла или каталога, NTFS возвращает код статуса повторного разбора, который сигнализирует драйверам фильтров файловой системы, подключенным к дисковому тому, и диспетчеру ввода-вывода о необходимости анализа данных повторного разбора. Каждый тип точек повторного разбора имеет уникальный **тэг повторного**

разбора (reparse tag) — он позволяет компоненту, отвечающему за интерпретацию данных конкретной точки повторного разбора, распознавать свои точки разбора, не проверяя их данные. Далее владелец тэга повторного разбора (драйвер фильтра файловой системы или диспетчер ввода-вывода) может выбрать один из следующих вариантов дальнейших действий.

- Владелец тэга повторного разбора может манипулировать полным именем файла, при анализе которого обнаружена точка повторного разбора, и инициировать ввод-вывод по измененному пути. Точки соединения используют этот вариант, например, для перенаправления каталогов.
- Владелец тэга повторного разбора может удалить из файла точку повторного разбора, каким-либо образом изменить файл, а затем инициировать новую операцию файлового ввода-вывода. По такому принципу точки повторного разбора используются системой Hierarchical Storage Management (HSM). HSM архивирует файлы, перемещая их содержимое на ленточные накопители и оставляя вместо содержимого файлов точки повторного разбора.

Сжатие и разреженные файлы

Приложения сжимают и разархивируют файлы, передавая DeviceIoControl управляющий код FSCTL_SET_COMPRESSION. Для запроса состояния сжатия файла или каталога используется управляющий код FSCTL_GET_COMPRES-SION. У сжатого файла или каталога установлен флаг FILE_ATTRIBUTE_COM-PRESSED, поэтому приложения могут определять состояние сжатия файла или каталога вызовом `GetFileAttributes`.

Второй тип сжатия известен под названием **разреженные файлы** (sparse files). Если файл помечен как разреженный, NTFS не выделяет на томе место для тех частей файла, которые определены приложением как пустые. При чтении приложением пустых областей разреженного файла NTFS просто возвращает буферы, заполненные нулевыми значениями. Этот тип сжатия полезен для клиент-серверных приложений, в которых реализовано протоколирование с циклическими буферами (circular-buffer logging): сервер регистрирует информацию в файле, а клиент асинхронно считывает ее.

Как и в случае сжатых файлов, NTFS прозрачно управляет разреженными файлами. Приложения указывают состояние разреженности файла, передавая DeviceIoControly правляющий код FSCTL_SET_SPARSE. Чтобы определить диапазон файла как пустой, приложения используют код FSCTL_SET_ 2ERO_DATA, а чтобы запросить у NTFS описание того, какие части файла являются разреженными, — код FSCTL_QUERY_ALLOCATED_RANGES.

Протоколирование изменений

Приложениям многих типов нужно отслеживать изменения файлов и каталогов тома. Например, программа автоматического резервного копирования первоначально выполняет полное резервное копирование, а в дальнейшем копирует только измененные файлы. Очевидный способ мониторинга изменений тома — его сканирование с записью состояния файлов и каталогов и анализ отличий при следующем сканировании. Однако этот процесс может негативно повлиять на производительность системы — особенно на компьютерах, хранящих тысячи и десятки тысяч файлов.

Альтернативный подход для приложения заключается в том, чтобы зарегистрироваться на получение уведомлений об изменении содержимого каталогов. Для этого предназначена Windows-функция `FindFirstChangeNotification` или `ReadDirectoryChangesW`. В качестве входного параметра приложение указывает имя нужного каталога, и функция сообщает о любом изменении в содержимом этого каталога. Хотя этот подход более эффективен, чем сканирование тома, он требует непрерывной работы приложения. При этом приложениям все равно может понадобиться сканирование каталогов, так как `FindFirstChangeNotification` сообщает лишь о факте изменений, а не о конкретных изменениях. В то же время `ReadDirectoryChangesW` принимает от приложения буфер, который FSD заполняет записями об изменениях. Но при переполнении буфера приложение должно быть готово вернуться к сканированию каталога.

NTFS предусматривает третий подход, в котором преодолены недостатки первых двух: приложение может настроить журнал изменений NTFS с помощью функции `DeviceIoControl` и управляющего кода FSCTL_CREATE_USNJOURNAL; тогда NTFS будет

регистрировать информацию об изменениях файлов и каталогов во внутреннем файле — **журнале изменений** (change journal). Для чтения записей в журнале изменений предназначен управляющий код FSCTL_QUERY_USNJOURNAL; при этом можно указать, чтобы функция `DeviceIoControl` не завершалась до тех пор, пока в журнале не появятся новые записи.

Отслеживание ссылок

Отслеживание связей в NTFS реализуется на основе необязательного атрибута файла, известного под названием идентификатор объекта (object ID). Приложение может назначить такой идентификатор файлу с помощью управляющих кодов файловой системы FSCTL_CREATE_OR_GET_OBJECT_ID (назначает идентификатор, если он еще не назначен) и FSCTL_SET_OBJECT_ID. Идентификаторы объектов можно запросить с помощью управляющих кодов FSCTL_CREATE_OR_GET_OBJECT_ID и FSCTL_GET_OBJECT_ID. Код FSCTL_DELETE_OBJECT_ID позволяет удалять идентификаторы объектов из файлов.

Шифрование

NTFS поддерживает механизм **Encrypting File System** (EFS), с помощью которого пользователи могут шифровать конфиденциальные данные. EFS, как и механизм сжатия файлов, полностью прозрачен для приложений. Это означает, что данные автоматически расшифровываются при чтении их приложением, работающим под учетной записью пользователя, который имеет права на просмотр этих данных, и автоматически шифруются при изменении их авторизованным приложением.

EFS использует криптографические сервисы, предоставляемые Windows в пользовательском режиме, и состоит из драйвера устройства режима ядра, тесно интегрированного с NTFS и DLL-модулями пользовательского режима, которые взаимодействуют с подсистемой локальной аутентификации (LSASS) и криптографическими DLL.

Доступ к зашифрованным файлам можно получить только с помощью закрытого ключа из криптографической пары EFS (которая состоит из закрытого и открытого ключей), а закрытые ключи защищены паролем учетной записи. Таким образом, без пароля учетной записи, авторизованной для просмотра данных, доступ к зашифрованным EFS файлам на потерянных или краденых портативных компьютерах нельзя получить никакими средствами (кроме грубого перебора паролей).

Windows-функции `EncryptFile` и `DecryptFile` позволяют шифровать и дешифровать файлы, а `FileEncryptionStatus` — получать атрибуты файла или каталога, связанного с EFS, — например, чтобы определить, зашифрован ли данный файл или каталог.

Драйвер файловой системы NTFS

NTFS и другие файловые системы представляют собой загружаемые драйверы устройств режима ядра. Они неявно вызываются приложениями, использующими Windows или другие API ввода-вывода (например, POSIX). Как показано на Рис. 11, подсистемы окружения вызывают системные сервисы, которые в свою очередь находят соответствующие загруженные драйверы и вызывают их.

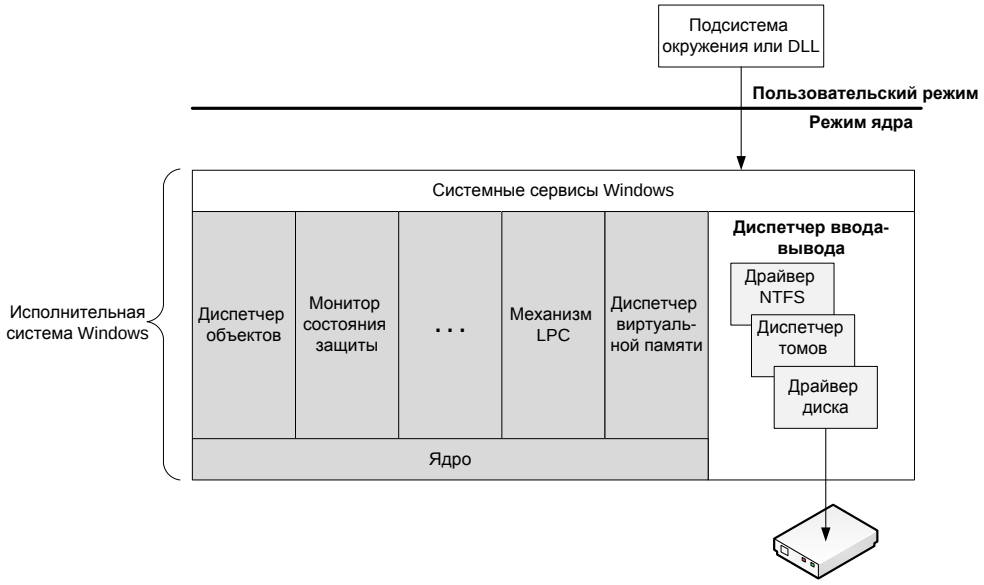


Рис. 11. Компоненты подсистемы ввода-вывода в Windows

Драйверы передают друг другу запросы ввода-вывода, вызывая диспетчер ввода-вывода исполнительной системы. Использование диспетчера ввода-вывода в качестве промежуточного звена обеспечивает независимость каждого драйвера, что позволяет загружать и выгружать его без последствий для других драйверов. Кроме того, драйвер NTFS взаимодействует с тремя другими компонентами исполнительной системы (Рис. 12), тесно связанными с файловыми системами.

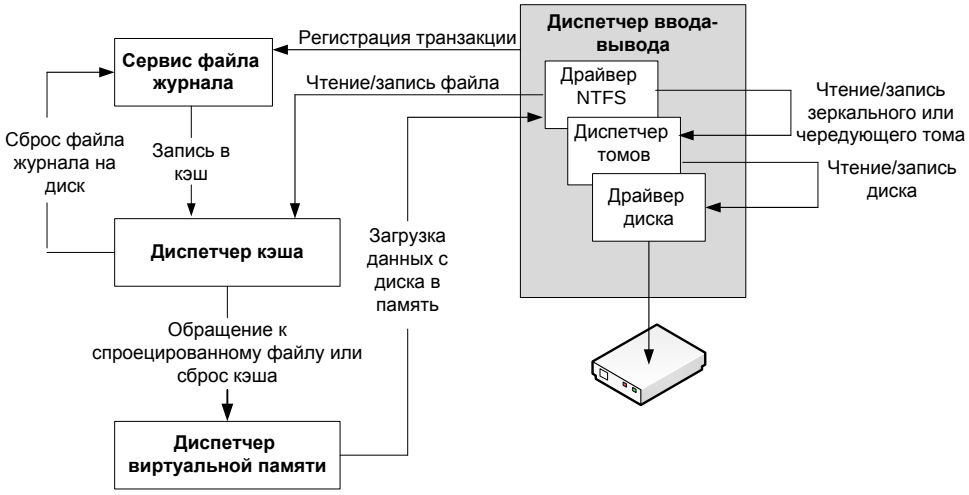


Рис. 12. NTFS и связные с ней компоненты

Сервис файла журнала (log file service, LFS) является частью NTFS и предоставляет функции для поддержки журнала изменений на диске. Файл журнала LFS используется при восстановлении тома NTFS в случае аварии системы (подробнее о LFS см. раздел «Сервис файла журнала» далее в этой главе).

Диспетчер кэша — компонент исполнительной системы, предоставляющий общесистемные сервисы кэширования для NTFS и драйверов других файловых систем, в том числе сетевых (т. е. для серверов и редиректоров). Все файловые системы, реализованные в Windows, получают доступ к кэшированным файлам, проецируя их на системное адресное пространство, а затем считывая соответствующие участки виртуальной памяти. С этой целью диспетчер кэша предоставляет диспетчеру памяти специализированный интерфейс файловых систем. Когда программа пытается обратиться к какой-либо части файла, не загруженной в кэш (промах кэша), диспетчер памяти вызывает NTFS для обращения к драйверу диска и загрузки нужных файловых данных. Диспетчер кэша оптимизирует дисковый ввод-вывод, вызывая диспетчер памяти (для сброса на диск содержимого кэша в фоновом режиме) из потоков подсистемы отложенной записи.

NTFS участвует в модели объектов Windows, реализуя файлы в виде объектов. Такая реализация допускает совместное использование файлов и их защиту диспетчером объектов, который управляет всеми объектами уровня исполнительной системы.

Приложение создает файлы и обращается к ним так же, как и к любым другим объектам Windows, — через описатели объектов. К тому времени, когда запрос ввода-вывода достигает NTFS, диспетчер объектов и система защиты уже убедились в наличии у вызывающего процесса прав на запрошенные им виды доступа к объекту «файл». Система защиты сравнила маркер доступа вызывающего процесса с элементами ACL этого объекта «файл», а диспетчер ввода-вывода преобразовал описатель файла в указатель на объект «файл». NTFS использует информацию из объекта «файл» для обращения к файлу на диске.

На Рис. 9 показаны структуры данных, связывающие описатель файла со структурой файловой системы на диске.

NTFS получает адрес файла на диске из объекта «файл» по нескольким указателям. Как видно из Рис. 9, объект «файл», представляющий один вызов системного сервиса для открытия файла, указывает на блок управления потоком данных (stream control block, SCB) для атрибута, который вызывающая программа пытается считать или записать. В нашем случае процесс открыл как безымянный атрибут данных, так и именованный поток (альтернативный атрибут данных) файла. SCB представляют отдельные атрибуты файла и содержат информацию о том, как искать конкретные атрибуты внутри файла. Все SCB файла указывают на общую структуру данных — блок управления файлом (file control block, FCB). FCB содержит указатель на запись файла в главной таблице файлов (MFT), о которой мы поговорим в следующем разделе.

Структура NTFS на диске

Тома

Структура NTFS начинается с тома. Том (volume) соответствует логическому разделу на диске и создается при форматировании диска или его части под NTFS. Оснастка Disk Management (Управление дисками) консоли MMC также позволяет создать том RAID, охватывающий несколько дисков.

На диске может быть один или несколько томов. NTFS обрабатывает каждый том независимо от других. Три примера конфигурации 150-мегабайтного жесткого диска показаны на Рис. 13.

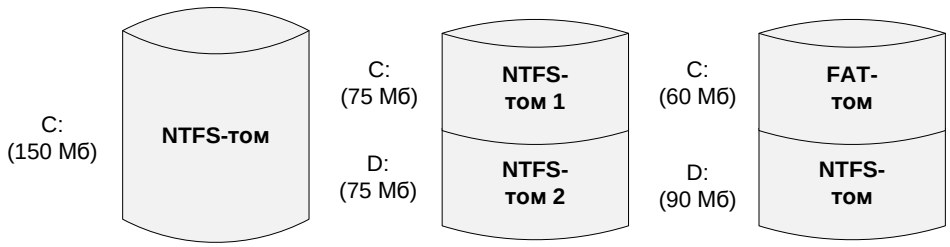


Рис. 13. Примеры конфигурации диска

Том состоит из набора файлов и свободного пространства, оставшегося в данном разделе диска. В FAT том также содержит области, специально отформатированные для использования файловой системой. Но в томе NTFS все данные файловой системы вроде битовых карт, каталогов и даже начального загрузочного кода хранятся как обычные файлы.

Кластеры

Размер кластера на томе NTFS, или **кластерный множитель** (cluster factor), устанавливается при форматировании тома командой `format` или в оснастке Disk Management (Управление дисками). Размер кластера по умолчанию определяется размером тома, но всегда содержит целое число физических секторов с дискретностью N2 (т. е. 1 сектор, 2 сектора, 4 сектора, 8 секторов и т. д.). Кластерный множитель выражается числом байтов в кластере, например 512 байт, 1 Кб или 2 Кб.

Внутренне NTFS работает только с кластерами. NTFS использует кластер как единицу выделения пространства для поддержания независимости от размера физического сектора. Это позволяет NTFS эффективно работать с очень большими дисками, используя кластеры большего размера, и поддерживать нестандартные диски с размером секторов, отличным от 512 байтов. Применение больших кластеров на больших томах уменьшает фрагментацию и ускоряет выделение свободного пространства за счет небольшого проигрыша в эффективности использования дискового пространства. Команда `format` или оснастка Disk Management выбирает кластерный множитель в зависимости от размера тома, но это значение можно изменить.

Главная таблица файлов

В NTFS все данные, хранящиеся на томе, содержатся в файлах. Это относится и к структурам данных, используемым для поиска и выборки файлов, к начальному загрузочному коду и битовой карте, в которой регистрируется состояние пространства всего тома (метаданные NTFS). Хранение всех видов данных в файлах позволяет файловой системе легко находить и поддерживать данные, а каждый файл может быть защищен дескриптором защиты. Кроме того, при появлении плохих секторов на диске, NTFS может переместить файлы метаданных.

Главная таблица файлов (MFT) занимает центральное место в структуре NTFS-тома. MFT реализована как массив записей о файлах. Размер каждой записи фиксирован и равен 1 Кб [3]. Логически MFT содержит по одной строке на каждый файл тома, включая строку для самой MFT. Кроме MFT, на каждом томе NTFS имеется набор файлов метаданных с информацией, необходимой для реализации структуры файловой системы. Имена всех файлов метаданных NTFS начинаются со знака доллара ($), хотя эти знаки скрыты. Так, имя файла MFT — $Mft. Остальные файлы NTFS-тома являются обычными файлами и каталогами (Рис. 14).

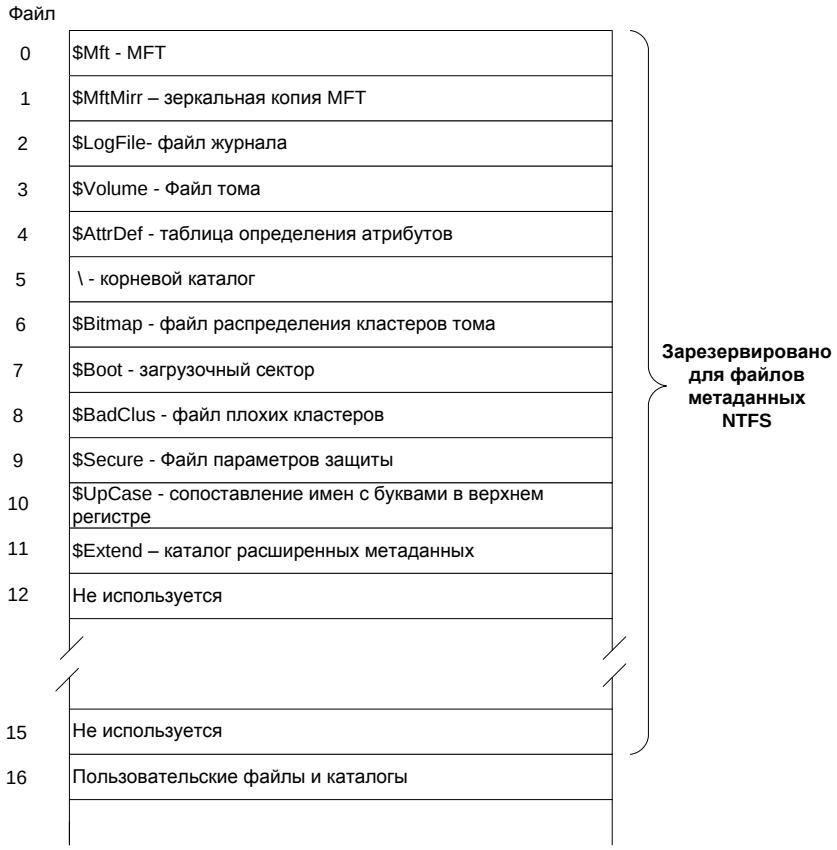


Рис. 14. Записи MFT для метаданных NTFS

Обычно каждая запись MFT соответствует отдельному файлу, но если у файла много атрибутов или он сильно фрагментирован, для него может понадобиться более одной записи. Тогда первая запись MFT, хранящая адреса других записей, называется **базовой** (base file record).

Файл журнала с именем **$LogFile:** NTFS использует его для регистрации всех операций, влияющих на структуру тома NTFS, в том числе для регистрации создания файлов и выполнения любых команд вроде Copy, модифицирующих структуру каталогов.

Элемент MFT зарезервирован для корневого каталога (также обозначаемого как **«\»**): его запись содержит индекс файлов и каталогов, хранящихся в корне структуры каталогов NTFS. Когда NTFS впервые получает запрос на открытие файла, она начинает его поиск с записи корневого каталога.

NTFS регистрирует распределение дискового пространства в файле битовой карты (bitmap file) с именем **$Bitmap**.

Файл защиты (security file) с именем **$Secure** хранит базу данных дескрипторов защиты, действующих в пределах тома. Дескрипторы защиты файлов и каталогов NTFS можно настраивать индивидуально, но для экономии места NTFS хранит настройки дескрипторов защиты в общем файле, который позволяет файлам и каталогам с одинаковыми параметрами защиты ссылаться на один и тот же дескриптор защиты.

Загрузочный файл (boot file), с именем **$Boot** хранит код начальной загрузки Windows. Для успешного запуска системы код начальной загрузки должен находиться на диске в определенном месте. При форматировании команда `format` определяет это место в виде файла, создавая для него запись в MFT.

Файл плохих кластеров (bad-cluster file) с именем **$BadClus**: в нем регистрируются все сбойные участки дискового тома, и файл тома (volume file) с именем $Volume, который содержит имя тома, версию NTFS, под которую отформатирован том, и бит, устанавливаемый при каком-либо повреждении диска. Если этот бит установлен, диск должен быть восстановлен утилитой Chkdsk. Файл сопоставления имен с буквами в верхнем регистре (uppercase file) с именем **$UpCase** включает таблицу трансляции букв между верхним и нижним регистрами. NTFS также поддерживает файл, содержащий таблицу определения атрибутов (attribute definition table), с именем **$AttrDef**; в нем определяются типы атрибутов, поддерживаемые томом, и указывается, являются ли они индексируемыми, следует ли их восстанавливать в ходе операции восстановления системы и т. д.

Некоторые файлы метаданных NTFS хранит в каталоге расширенных метаданных **$Extend**, в том числе помещая туда файл идентификаторов объектов ($ObjId), файл квот ($Quota), файл журнала регистрации изменений ($UsnJrnl) и файл точек повторного разбора ($Reparse).

Имена файлов

NTFS и FAT допускают длину каждого имени файла в пути до 255 символов. Эти имена могут содержать Unicode-символы, точки и пробелы. Однако длина имен файлов в FAT, встроенной в MS-DOS, ограничена 8 символами (не-Unicode), за которыми следует расширение из трех символов, отделенное точкой. Рис. 15 иллюстрирует различные пространства имен файлов, поддерживаемые Windows, и показывает, как они перекрываются.

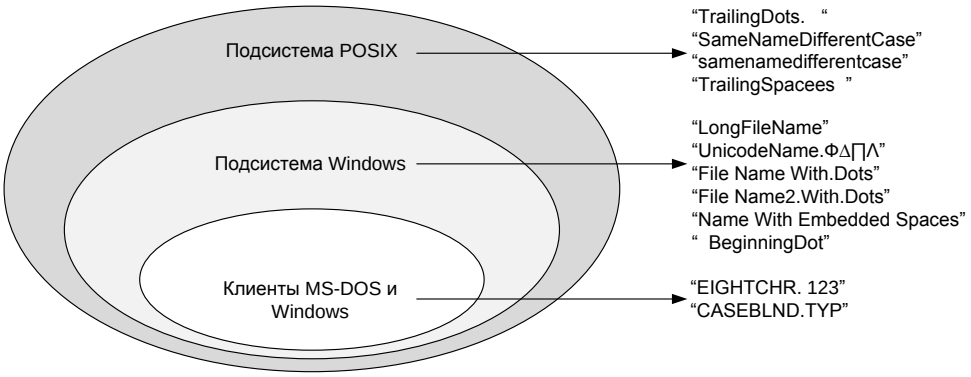


Рис. 15. Пространства имен файлов, поддерживаемые Windows

POSIX требует самого большого пространства имен из всех подсистем, поддерживаемых Windows. Подсистема POSIX может создавать имена, невидимые приложениям Windows и MS-DOS, в том числе имена с концевыми точками и концевыми пробелами. Создание файла в большом пространстве имен POSIX обычно не является проблемой, потому что

вы создаете такой файл для использования подсистемой POSIX или ее клиентской системой.

Но взаимосвязь между 32-разрядными Windows-приложениями и программами MS-DOS и 16-разрядной Windows намного теснее. Область, отведенная Windows, представляет имена файлов, которые подсистема Windows может создавать на томе NTFS, хотя такие имена невидимы программам MS-DOS и 16-разрядной Windows. В эту группу входят имена файлов, не соответствующие формату «8.3».

Имена MS-DOS — полнофункциональные псевдонимы файлов NTFS и хранятся в том же каталоге, что и длинные имена. На Рис. 16 показана запись MFT для файла с автоматически сгенерированным MS-DOS-именем.

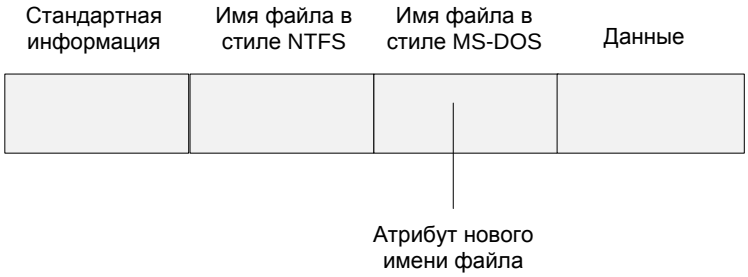


Рис. 16. Запись MFT с атрибутом имени файла для MS-DOS

Имя NTFS и сгенерированное имя MS-DOS хранятся в той же записи и относятся к одному и тому же файлу. Имя MS-DOS можно использовать для открытия, чтения, записи и копирования файла. Если пользователь переименовывает файл, заменяя длинное имя на краткое или наоборот, новое имя заменяет оба существовавших варианта. Если новое имя является недопустимым для MS-DOS, NTFS генерирует для файла другое MS-DOS-имя.

Вот алгоритм, применяемый NTFS при генерации краткого MS-DOS-имени из длинного.

1. Удалить из длинного имени все символы, недопустимые в именах MS-DOS, включая пробелы и Unicode-символы. Удалить начальную и концевую точки, а также все внутренние точки, кроме последней.

2. Урезать часть строки перед точкой (если она есть) до шести символов и добавить строку «~n» (где n —порядковый номер, который начинается с 1; он нужен, чтобы различать файлы, урезание имен которых дает одинаковый результат). Урезать строку после точки (если она есть) до трех символов.

3. Преобразовать полученный набор символов в верхний регистр. MS-DOS нечувствительна к регистру букв в именах файлов, но эта операция гарантирует, что NTFS не сгенерирует для файла новое имя, отличающееся от старого лишь регистром.

4. Если сгенерированное имя дублирует уже имеющееся в каталоге, увеличить порядковый номер в строке «~n» на 1 (или на большее значение).

Резидентные и нерезидентные атрибуты

Если файл невелик, все его атрибуты и их значения (например, файловые данные) умещаются в одной записи файла. Когда значение атрибута хранится непосредственно в MFT, атрибут называется **резидентный** (например, все атрибуты на Рис. 15 являются резидентными). Некоторые атрибуты всегда резидентны — по ним NTFS находит нерезидентные атрибуты. Так, атрибуты «стандартная информация» и «корень индекса» всегда резидентны.

Каждый атрибут начинается со стандартного заголовка, в котором содержится информация об атрибуте, используемая NTFS для базового управления атрибутами. В заголовке, который всегда является резидентным, регистрируется, резидентно ли значение данного атрибута. В случае резидентных атрибутов заголовок также содержит смещение значения атрибута от начала заголовка и длину этого значения (Рис. 17).

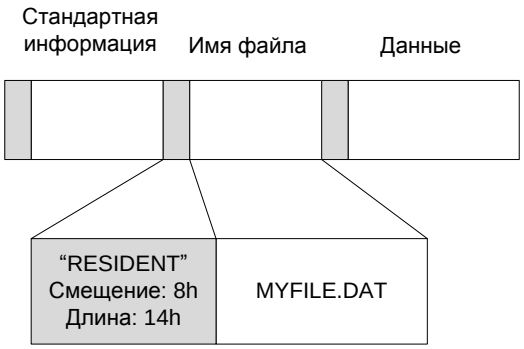


Рис. 17. Заголовок и значение резидентного атрибута

Когда значение атрибута хранится непосредственно в MFT, обращение к нему занимает значительно меньше времени. Вместо того чтобы искать файл в таблице, а затем считывать цепочку кластеров для поиска файловых данных (как, например, поступает FAT), NTFS обращается к диску только один раз и немедленно считывает данные.

Как видно из Рис. 18, атрибуты небольшого каталога, а также небольшого файла, могут быть резидентными в MFT. Для небольшого каталога атрибут «корень индекса» содержит индекс файловых ссылок на файлы и подкаталоги этого каталога.

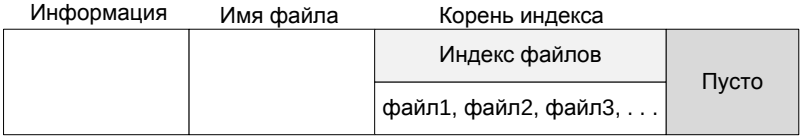


Рис. 18. Запись в MFT для небольшого каталога

Конечно, многие файлы и каталоги нельзя втиснуть в запись MFT с фиксированным размером в 1 Кб. Если некий атрибут, например файловые данные, слишком велик и не умещается в записи MFT, NTFS выделяет для него отдельные кластеры за пределами MFT. Эта область называется **группой** (run) или **экстентом** (extent). Если размер значения атрибута впоследствии расширяется (например, при добавлении в файл дополнительных данных), NTFS выделяет для новых данных еще одну группу. Атрибуты, значения которых хранятся в группах, а не в MFT, называются **нерезидентными**. Файловая система сама решает, будет атрибут резидентным или нерезидентным, и обеспечивает пользовательским процессам прозрачный доступ к этим данным.

В случае нерезидентного атрибута (им может быть атрибут данных большого файла) в его заголовке содержится информация, нужная NTFS для поиска значения атрибута на диске. На Рис. 19 показан нерезидентный атрибут данных, хранящийся в двух группах.

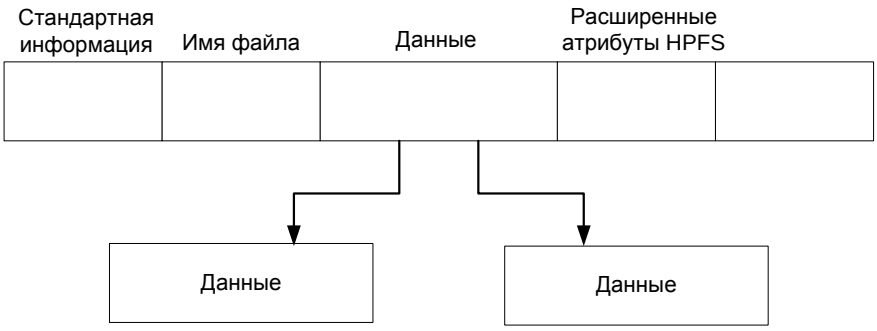


Рис. 19. Запись в MFT для большого файла с двумя группами данных

В большом каталоге также могут быть нерезидентные атрибуты (или части атрибутов), как видно из Рис. 20. В этом примере в записи MFT не хватает места для хранения индекса файлов, составляющих этот большой каталог. Часть индекса хранится в атрибуте корня индекса, а остальное — в нерезидентных группах, называемых **индексными буферами** (index buffers). Атрибуты корня индекса, выделенной группы

индексов (index allocation) и битовой карты показаны здесь в упрощенной форме. Атрибуты стандартной информации и имени файла всегда резидентны. Заголовок и по крайней мере часть значения атрибута корня индекса в случае каталогов также резидентны.

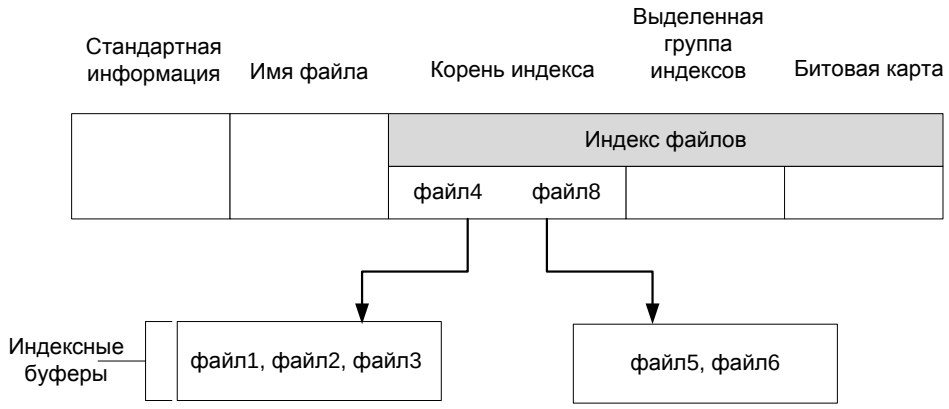


Рис. 20. Запись MFT для большого каталога с нерезидентным индексом имен файлов

Когда атрибуты файла (или каталога) не умещаются в записи MFT и для них требуется отдельное место, NTFS отслеживает выделяемые группы посредством пар сопоставлений VCN-LCN. LCN представляют последовательность кластеров на всем томе, пронумерованных от 0 до n. VCN нумеруют от 0 до m только кластеры, принадлежащие конкретному файлу. Пример нумерации кластеров в группах нерезидентного атрибута данных приведен на Рис. 21.

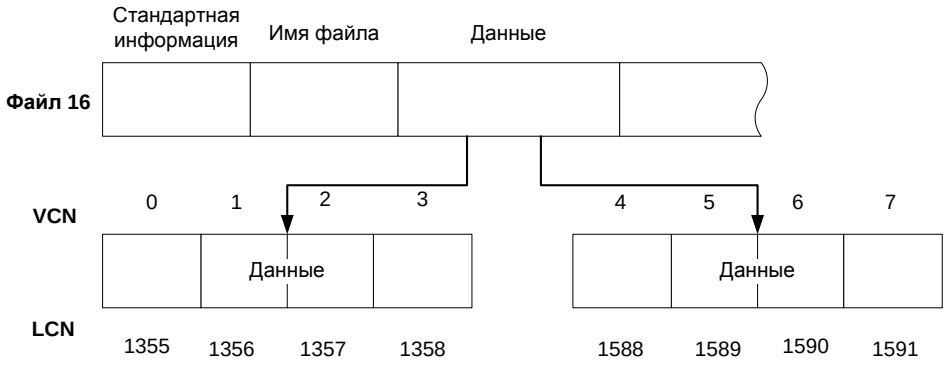


Рис. 21. VCN для нерезидентного атрибута данных

Если бы этот файл занимал больше двух групп, нумерация в третьей группе началась бы с VCN 8. Как показано на Рис. 22, заголовок атрибута данных содержит сопоставления VCN-LCN для обоих групп, что позволяет NTFS легко находить выделенные под них области на диске.

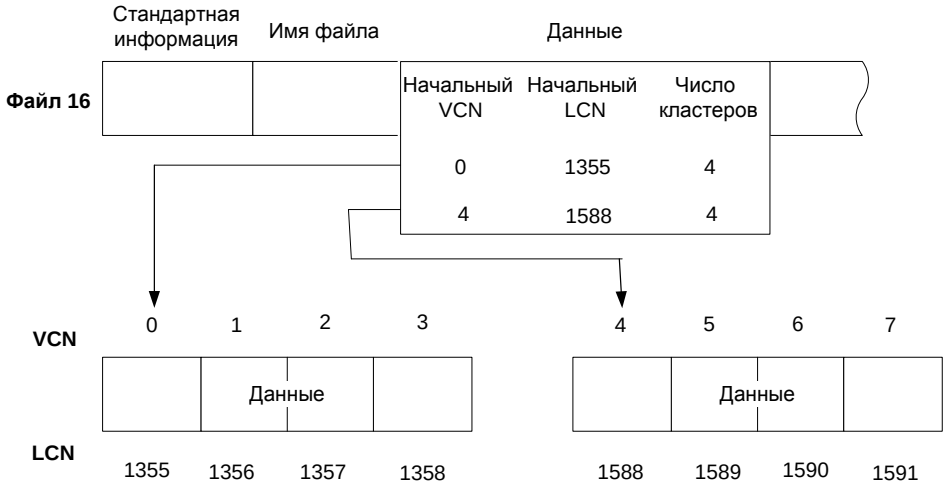


Рис. 22. Сопоставления VCN-LCN для нерезидентного атрибута данных

Сжатие данных и разреженные файлы

NTFS поддерживает сжатие по отдельным файлам, по каталогам и по томам (NTFS сжимает только пользовательские данные, не трогая метаданные файловой системы). Выяснить, сжат ли том, можно через Windows-функцию `GetVolumeInformation`. Чтобы получить реальный размер сжатого файла, используйте Windows-функцию `GetCompressedFileSize`. Наконец, проверить или изменить параметры сжатия для файла или каталога позволяет Windows-функция `DeviceIoControl` (см. управляющие коды файловой системы FSCTL_ GET_COMPRESSION и FSCT_SET_COMPRESSION в описании этой функции в Platform SDK). Учтите, что изменение степени сжатия применительно к файлу выполняется немедленно, а применительно к каталогу или тому — нет. Во втором случае степень сжатия, заданная для каталога или тома, становится степенью сжатия по умолчанию для всех новых файлов и подкаталогов, создаваемых в каталоге или на томе.

Поддержка восстановления в NTFS

Поддержка восстановления в NTFS гарантирует, что в случае отказа электропитания или аварии системы ни одна операция файловой системы (транзакция) не останется незавершенной; при этом структура дискового тома будет сохранена. NTFS включает утилиту Chkdsk, которая позволяет устранять последствия катастрофических повреждений диска, вызванных аппаратными ошибками ввода-вывода (например, из-за аварийных секторов на диске, электрических аномалий или сбоев в работе диска) либо ошибками в программном обеспечении. Наличие средств восстановления NTFS уменьшает потребность в использовании Chkdsk.

Восстанавливаемые файловые системы

Восстанавливаемая файловая система типа NTFS превосходит по надежности файловые системы с точной записью и при этом достигает уровня производительности файловых систем с отложенной записью.

Восстанавливаемая файловая система NTFS использует стратегию, повышающую ее надежность по сравнению с традиционными файловыми системами. **Во-первых**, восстанавливаемость NTFS гарантирует, что структура тома не будет разрушена, так что в случае сбоя системы все файлы останутся доступными.

Во-вторых, хотя NTFS не гарантирует сохранности пользовательских данных в случае сбоя системы (некоторые изменения в кэше могут быть потеряны), приложения могут использовать преимущества сквозной записи и сброса кэша NTFS для гарантии того, что изменения файлов будут записываться на диск в должное время. NTFS не требуется дополнительного ввода-вывода для сброса на диск изменений нескольких различных структур данных файловой системы, так как изменения в этих структурах регистрируются в файле журнала (в ходе единственной операции записи); если произошел сбой и содержимое кэша потеряно, изменения файловой системы могут быть восстановлены по информации из журнала. Более того, NTFS в отличие от FAT гарантирует, что сразу после операции сквозной записи или сброса кэша пользовательские данные останутся целостными и будут доступны, даже если вслед за этим произойдет сбой системы.

Сервис файла журнала

Сервис файла журнала (log file service, LFS) — это набор процедур режима ядра, локализованных в драйвере NTFS, который она использует для доступа к файлу журнала. Хотя LFS изначально был разработан для того, чтобы предоставлять средства протоколирования и восстановления более чем одному клиенту, он используется только NTFS. Вызывающая программа, в данном случае NTFS, передает LFS указатель на открытый объект «файл», который определяет файл, выступающий в роли журнала. LFS либо инициализирует новый журнал, либо вызывает диспетчер кэша для доступа к существующему журналу через кэш, как показано на Рис. 23.

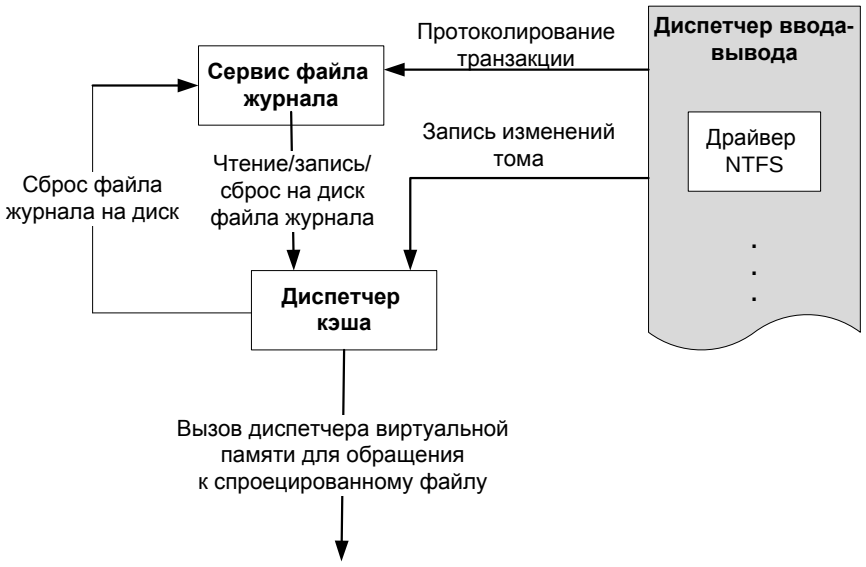


Рис. 23. Сервис файла журнала (LFS)

LFS делит файл журнала на две части: область перезапуска (restart area) и «безразмерную» область протоколирования (logging area).

NTFS вызывает LFS для чтения и записи области перезапуска. В этой области NTFS хранит информацию о контексте, например позицию в области протоколирования, откуда начнется чтение при восстановлении после сбоя системы. На тот случай, если область перезапуска будет разрушена или по каким-либо причинам станет недоступной, LFS создает ее копию. Остальная часть журнала транзакций — это область протоколирования, в которой находятся записи транзакций, обеспечивающие восстановление тома после сбоя. LFS создает иллюзию бесконечности журнала транзакций за счет его циклического повторного использования (в то же время не перезаписывая нужную информацию). Для идентификации записей, помещенных в журнал, LFS использует **номера логической последовательности** (logical sequence number, LSN). Циклически используя журнал, LFS увеличивает значения LSN.

Вот как система обеспечивает восстановление тома.

1. Сначала NTFS вызывает LFS для записи в кэшируемый файл журнала любых транзакций, модифицирующих структуру тома.

2. NTFS модифицирует том (также в кэше).

3. Диспетчер кэша сообщает LFS сбросить файл журнала на диск (Рис. 23).

4. Сбросив на диск журнал транзакций, диспетчер кэша записывает на диск изменения тома (результаты операций над метаданными). Эта последовательность действий гарантирует: если завершить изменение файловой системы не удастся, соответствующие транзакции можно будет считать из журнала и либо повторить, либо отменить в процессе восстановления файловой системы.

Восстановление файловой системы начинается автоматически при первом обращении к дисковому тому после перезагрузки. NTFS проверяет, применялись ли к тому транзакции, запротоколированные в журнале до сбоя, и, если нет, повторяет их. NTFS гарантирует и отмену транзакций, не полностью запротоколированных до сбоя, так что вызываемые ими изменения не появятся на томе.

Восстановление

NTFS автоматически выполняет восстановление диска при первом обращении к нему какой-либо программы после загрузки системы. (Если восстановление не требуется, весь процесс тривиален.) При восстановлении используются две таблицы, которые NTFS поддерживает в памяти.

- **Таблица транзакций** (transaction table) — предназначена для отслеживания начатых, но еще не зафиксированных транзакций. В процессе восстановления результаты подопераций этих транзакций должны быть удалены с диска.
- **Таблица измененных страниц** (dirty page table) — в нее записывается информация о том, какие страницы кэша содержат изменения структуры

файловой системы, еще не записанные на диск. Эти данные в процессе восстановления должны быть сброшены на диск.

Каждые 5 секунд NTFS добавляет в файл журнала транзакций запись контрольной точки. Непосредственно перед этим она обращается к LFS для сохранения в журнале текущей копии таблицы транзакций и таблицы измененных страниц. Затем NTFS запоминает в записи контрольной точки LSN записей журнала, содержащих копии таблиц. В начале процесса восстановления после сбоя NTFS обращается к LFS для поиска записей журнала транзакций, содержащих самую последнюю запись контрольной точки и самые последние копии упомянутых выше таблиц. Затем NTFS копирует эти таблицы в память.

При восстановлении тома NTFS выполняет три прохода по журналу транзакций, загружая его в память на первом проходе, чтобы минимизировать объем дискового ввода-вывода. Каждый проход имеет свое назначение:

- анализ;
- повтор транзакций;
- отмена транзакций.

Проход анализа

При **проходе анализа** (analysis pass) NTFS просматривает журнал транзакций в прямом направлении, начиная с последней операции контрольной точки, чтобы найти записи модификации и обновить скопированные ранее в память таблицы транзакций и измененных страниц. Обратите внимание, что операция контрольной точки помещает в журнал транзакций три записи, между которыми могут оказаться записи модификации (Рис. 24). NTFS должна приступить к сканированию с начала операции контрольной точки.

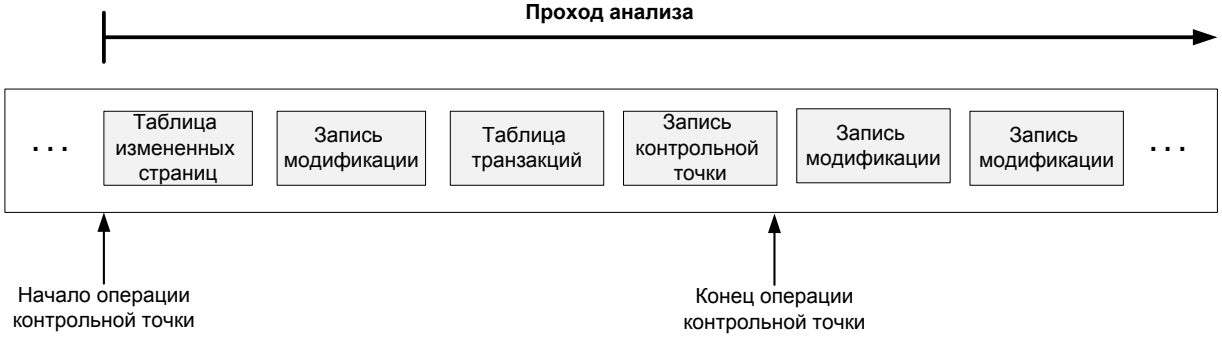


Рис. 24. Проход анализа

Большинство записей модификации, расположенных в журнале после начала операции контрольной точки, представляет собой изменение либо таблицы транзакций, либо таблицы измененных страниц.

После того как таблицы в памяти приведены в актуальное состояние, NTFS просматривает их, чтобы определить LSN самой старой записи модификации, которая регистрирует операцию, не выполненную над диском. Таблица транзакций содержит LSN незафиксированных (незавершенных) транзакций, а таблица измененных страниц — LSN записей, соответствующих модификациям кэша, не отраженным на диске. LSN самой старой записи, найденной NTFS в этих двух таблицах, определяет, откуда начнется проход повтора. Однако, если последняя запись контрольной точки окажется более ранней, NTFS начнет проход повтора именно с нее.

Проход повтора

На **проходе повтора** (redo pass) NTFS сканирует журнал транзакций в прямом направлении, начиная с LSN самой старой записи, которая была обнаружена на проходе анализа (Рис. 25). Она ищет записи модификации, относящиеся к обновлению страницы и содержащие модификации тома, которые были запротоколированы до сбоя системы, но не сброшены из кэша на диск. NTFS повторяет эти обновления в кэше.

Рис. 25. Проход повтора

Когда NTFS достигает конца журнала транзакций, она уже обновила кэш необходимыми модификациями тома, и подсистема отложенной записи, принадлежащая диспетчеру кэша, может начать переписывать содержимое кэша на диск в фоновом режиме.

Проход отмены

Завершив проход повтора, NTFS начинает проход отмены (undo pass), откатывая транзакции, не зафиксированные к моменту сбоя системы. На Рис. 26 показаны две транзакции в журнале: транзакция 1 зафиксирована до сбоя системы, а транзакция 2 — нет. NTFS должна отменить транзакцию 2.

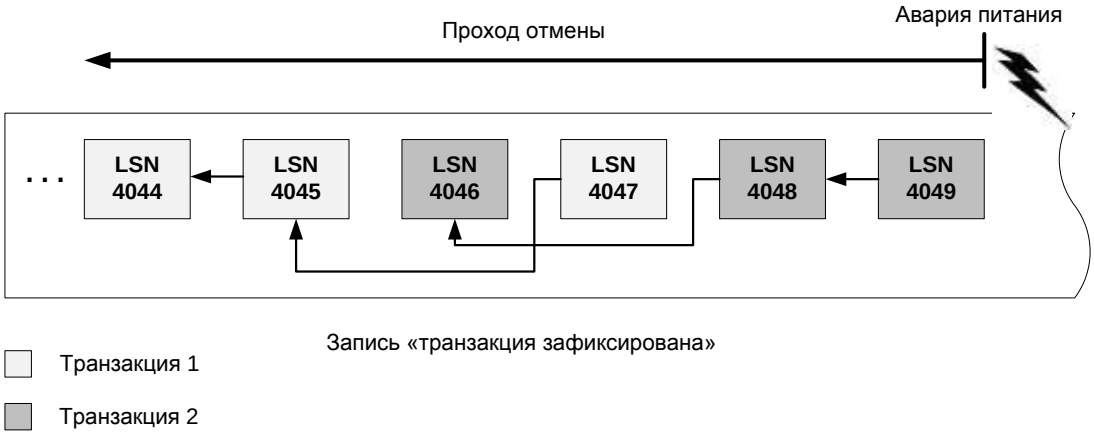


Рис. 26. Проход отмены

Когда проход отмены завершен, том возвращается в согласованное состояние. В этот момент NTFS сбрасывает на диск изменения кэша, чтобы гарантировать правильность содержимого тома. Далее NTFS записывает «пустую» область перезапуска, указывающую, что том находится в согласованном состоянии и что, если система сразу потерпит еще одну аварию, никакого восстановления не потребуется. На этом восстановление заканчивается.

NTFS гарантирует, что восстановление вернет том в некое существовавшее ранее целостное состояние, но не обязательно в непосредственно предшествовавшее сбою. NTFS не может дать такой гарантии, поскольку для большей производительности она использует алгоритм отложенной фиксации (lazy commit), а значит, сброс журнала транзакций из кэша на диск не выполняется немедленно всякий раз, когда добавляется запись «транзакция зафиксирована». Вместо этого несколько записей фиксации транзакций объединяются в пакет и записываются совместно — либо когда диспетчер кэша вызывает LFS для сброса журнала на диск, либо когда LFS помещает в журнал новую запись контрольной точки (каждые 5 секунд). Другая причина, по которой том не всегда возвращается к самому последнему состоянию, — в момент сбоя системы могли быть активны несколько параллельных транзакций, и одни записи фиксации этих транзакций были перенесены на диск, а другие — нет. Согласованное состояние тома, полученное в результате восстановления, отражает только те транзакции, чьи записи фиксации успели попасть на диск.

Восстановление плохих кластеров в NTFS

Диспетчеры томов Windows — FtDisk (для базовых дисков) и LDM (для динамических) — могут восстанавливать данные из плохого (аварийного) сектора на отказоустойчивом томе, но, если жесткий диск не является SCSI-диском или если на нем больше нет резервных секторов, они не в состоянии заменить плохой сектор новым. В таком случае, когда файловая система считывает данные из плохого сектора, диспетчер томов восстанавливает информацию и возвращает ее вместе с соответствующим предупреждением.

NTFS-эквивалент механизма замены секторов динамически заменяет кластер, содержащий плохой сектор, и ведет учет плохих кластеров, чтобы предотвратить их дальнейшее использование. Эта функциональность NTFS активизируется, если диспетчер томов не может заменить плохой сектор. Когда FtDisk возвращает предупреждение о плохом секторе или когда драйвер диска сообщает об ошибке, связанной с плохим сектором, NTFS выделяет новый кластер для замены того, который содержит плохой сектор. NTFS копирует данные, восстановленные диспетчером томов, в новый кластер, чтобы снова добиться избыточности данных.

На Рис. 27 показана запись MFT для пользовательского файла, в одном из групп которого имеется плохой сектор. Когда NTFS получает ошибку, связанную с плохим сектором, она присоединяет содержащий его кластер к своему файлу плохих кластеров. Это предотвращает повторное выделение данного кластера другому файлу. Затем NTFS выделяет новый кластер и изменяет сопоставление VCN-LCN для файла так, чтобы оно указывало на этот кластер. Данная процедура, известная как **переназначение плохого кластера** (bad-cluster remapping), иллюстрируется на Рис. 28. Кластер номер 1357, содержащий плохой сектор, заменяется новым кластером с номером 1049.

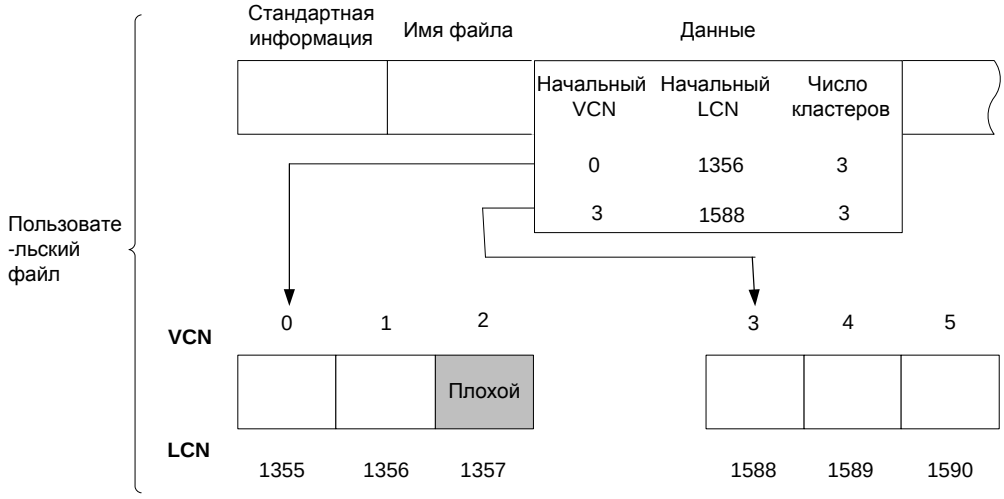
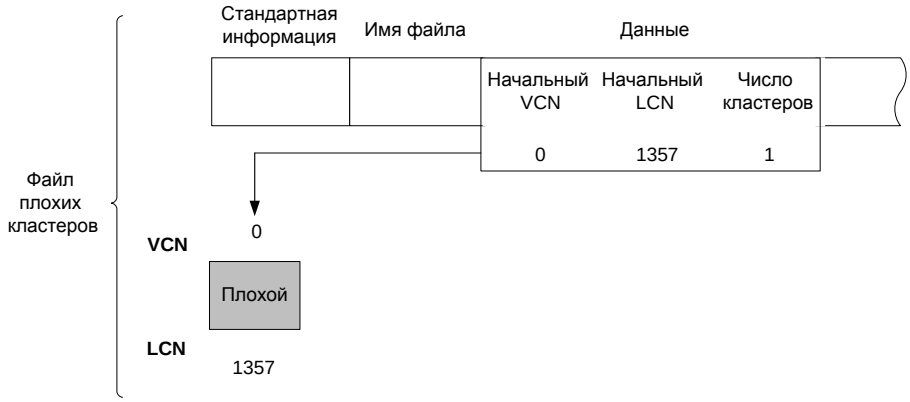


Рис. 27. Запись MFT для пользовательского файла с плохим кластером



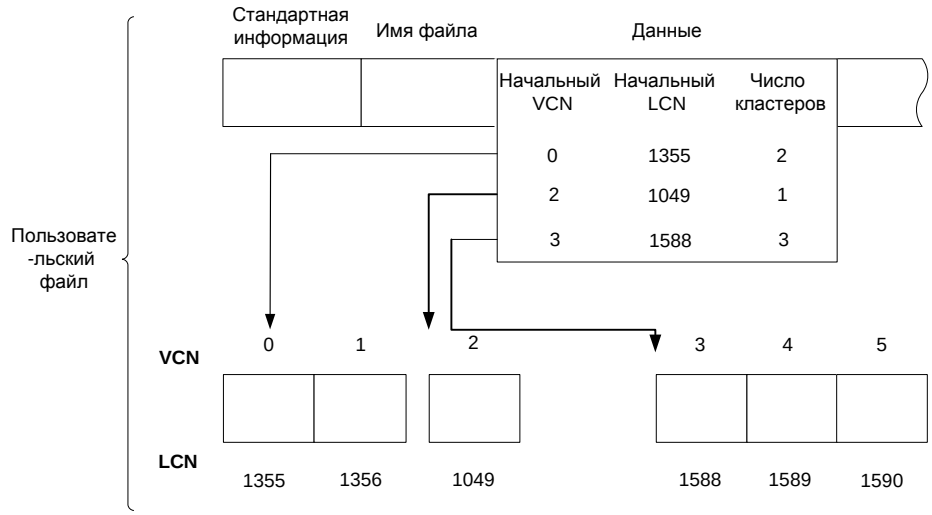


Рис. 28. Переназначение плохих кластеров

Механизм EFS

EFS (Encrypting File System) использует средства поддержки шифрования. При первом шифровании файла EFS назначает учетной записи пользователя, шифрующего этот файл, криптографическую пару — закрытый и открытый ключи. Пользователи могут шифровать файлы с помощью Windows Explorer; для этого нужно открыть диалоговое окно Properties (Свойства) применительно к нужному файлу, щелкнуть кнопку Advanced (Другие) и установить флажок Encrypt Contents To Secure Data (Шифровать содержимое для защиты данных), как показано на Рис. 29. Пользователи также могут шифровать файлы с помощью утилиты командной строки `cipher`. Windows автоматически шифрует файлы в каталогах, помеченных зашифрованными. При шифровании файла EFS генерирует случайное число, называемое шифровальным ключом файла (file encryption key, FEK). EFS использует FEK для шифрования содержимого файла по более стойкому варианту DES (Data Encryption Standard) — DESX (в Windows 2000), а также по DESX, 3DES (Triple-DES) или AES (Advanced Encryption Standard) в Windows XP (Service Pack 1 и выше) и Windows Server 2003. EFS сохраняет FEK вместе с самим файлом, но FEK шифруется по алгоритму RSA-шифрования на основе открытого ключа. После выполнения EFS этих действий файл защищен: другие пользователи не смогут расшифровать данные без расшифрованного FEK файла, а FEK они не смогут расшифровать без закрытого ключа пользователя — владельца файла.

Стойкость алгоритмов шифрования FEK

По умолчанию FEK шифруется в Windows 2000 и Windows XP по алгоритму DESX, а в Windows XP с Service Pack 1 (или выше) и Windows Server 2003 — по алгоритму AES. В версиях Windows, разрешенных к экспорту за пределы США, драйвер EFS реализует 56-битный ключ шифрования DESX, тогда как в версии, подлежаыщей использованию только в США, и в версиях с пакетом для 128-битного шифрования длина ключа DESX равна 128 битам. Алгоритм AES в Windows использует 256-битные ключи. Применение 3DES разрешает доступ к более длинным ключам, поэтому, если вам требуется более высокая стойкость FEK, вы можете включить шифрование 3DES одним из двух способов: как алгоритм шифрования для всех криптографических сервисов в системе или только для EFS.

Чтобы 3DES стал алгоритмом шифрования для всех системных криптографических сервисов, запустите редактор локальной политики безопасности, введя secpol.msc в диалоговом окне Run (Запуск программы), и откройте узел Security Options (Параметры безопасности) под Local Policies (Локальные политики). Найдите параметр System Cryptography: Use FIPS Compliant Algorithms For Encryption, Hashing And Signing (Системная криптография: использовать FIPS-совмести-мые алгоритмы для шифрования, хеширования и подписывания) и включите его.

Чтобы активизировать 3DES только для EFS, создайте DWORD-параметр HKLM\SOFWARE\Microsoft\Windows NT\CurrentVersion\EFS\ AlgorithmID, присвойте ему значение 0x6603 и перезагрузите систему.

Рис. 29. Шифрование файлов через пользовательский интерфейс Windows Explorer

Для шифрования FEK используется алгоритм криптографической пары, а для шифрования файловых данных — DESX, AES или 3DES. Как правило, алгоритмы симметричного шифрования работают очень быстро, что делает их подходящими для шифрования больших объемов данных, в частности файловых. Однако у алгоритмов симметричного шифрования есть одна слабая сторона: зашифрованный ими файл можно вскрыть, получив ключ. Если несколько человек собирается пользоваться одним файлом, защищенным только DESX, AES или 3DES, каждому из них понадобится доступ к FEK файла. Очевидно, что незашифрованный FEK — серьезная угроза безопасности. Но шифрование FEK все равно не решает проблему, поскольку в этом случае нескольким людям приходится пользоваться одним и тем же ключом расшифровки FEK.

Защита FEK - сложная проблема, для решения которой EFS использует ту часть своей криптографической архитектуры, которая опирается на технологии шифрования с открытым ключом. Шифрование FEK на индивидуальной основе позволяет нескольким лицам совместно использовать зашифрованный файл. EFS может зашифровать FEK файла с помощью открытого ключа каждого пользователя и хранить их FEK вместе с файлом. Каждый может получить доступ к открытому ключу пользователя, но никто не сможет расшифровать с его помощью данные, зашифрованные по этому ключу. Единственный способ расшифровки файла заключается в использовании операционной системой закрытого ключа. Закрытый ключ помогает расшифровать нужный зашифрованный экземпляр FEK файла. Алгоритмы на основе открытого ключа обычно довольно медленные, поэтому они используются EFS только для шифрования FEK. Разделение ключей на открытый и закрытый немного упрощает управление ключами по сравнению с таковым в алгоритмах симметричного шифрования и решает дилемму, связанную с защитой FEK.

Windows хранит закрытые ключи в подкаталоге Application Data\Micro-soft\Crypto\RSA каталога профиля пользователя. Для защиты закрытых ключей Windows шифрует все файлы в папке RSA на основе симметричного ключа, генерируемого случайным образом; такой ключ называется **мастер-ключом** пользователя. Мастер-ключ имеет длину в 64 байта и создается стойким генератором случайных чисел. Мастер-ключ также хранится в профиле пользователя в каталоге Application Data\Microsoft\Protect и зашифровывается по алгоритму 3DES с помощью ключа, который отчасти основан на пароле пользователя. Когда пользователь меняет свой пароль, мастер-ключи автоматически расшифровываются, а затем заново зашифровываются с учетом нового пароля.

Функциональность EFS опирается на несколько компонентов, как видно на схеме архитектуры EFS (Рис. 30). Всякий раз, когда NTFS встречает шифрованный файл, она вызывает функции EFS, зарегистрированные кодом EFS режима ядра в NTFS при инициализации этого кода. Функции EFS осуществляют шифрование и расшифровку файловых данных по мере обращения приложений к шифрованным файлам. Хотя EFS хранит FEK вместе с данными файла, FEK шифруется с помощью открытого ключа индивидуального пользователя. Для шифрования или расшифровки файловых данных

EFS должна расшифровать FEK файла, обращаясь к криптографическим сервисам пользовательского режима.

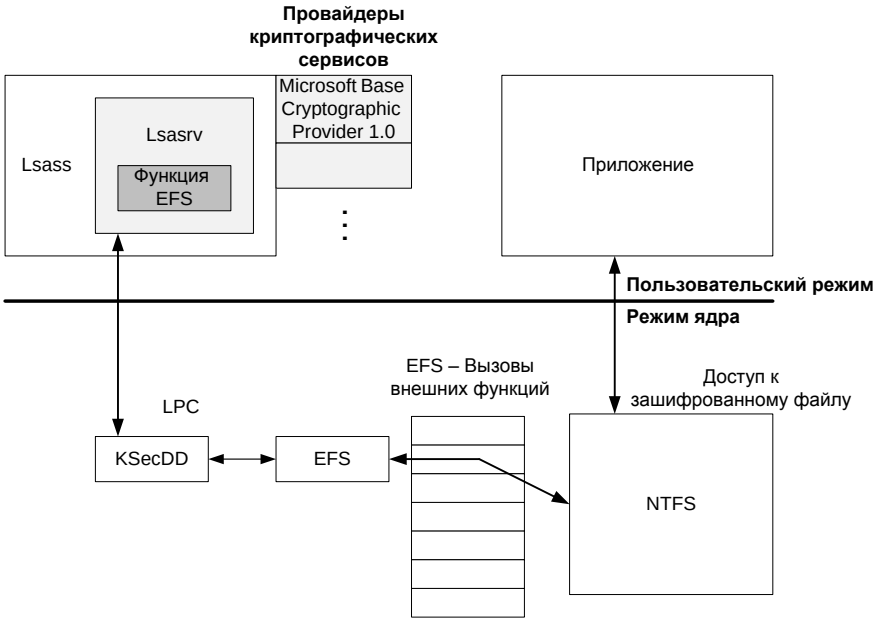


Рис. 30. Архитектура EFS

Подсистема локальной аутентификации (Local Security Authentication Subsystem, LSASS) (\Windows\System32\Lsass.exe) не только управляет сеансами регистрации, но и выполняет все рутинные операции, связанные с управлением ключами EFS. Драйвер устройства KSecDD (\Windows\System32\Drivers\Ksecdd.sys) экспортирует функции, необходимые драйверам для посылки LPC-сообщений LSASS. Компонент LSASS, сервер локальной аутентификации (Local Security Authentication Server, Lsasrv) (\Windows\System32\Lsasrv.dll), ожидает запросы на расшифровку FEK через RPC; расшифровка выполняется соответствующей функцией EFS, которая также находится в Lsasrv. Для расшифровки FEK, передаваемого LSASS драйвером EFS в зашифрованном виде, Lsasrv использует функции `Microsoft CryptoAPI (CAPI)`.

Первое шифрование файла

Обнаружив шифрованный файл, драйвер NTFS вызывает функции EFS. NTFS и EFS имеют специальные интерфейсы для преобразования файла из незашифрованной в зашифрованную форму, но этот процесс протекает в основном под управлением компонентов пользовательского режима.

Получив RPC-сообщение с запросом на шифрование файла от Feclient, Lsasrv использует механизм олицетворения Windows для подмены собой пользователя, запустившего программу, шифрующую файл. Это заставляет Windows воспринимать файловые операции, выполняемые Lsasrv, как операции, выполняемые пользователем, желающим зашифровать файл. Lsasrv обычно работает под учетной записью System.

Далее Lsasrv создает файл журнала в каталоге System Volume Information, где регистрирует ход процесса шифрования. Имя файла журнала — обычно EfsO.log, но, если шифруется несколько файлов, O заменяется числом, которое последовательно увеличивается на 1 до тех пор, пока не будет получено уникальное имя журнала для текущего шифруемого файла.

CryptoAPI полагается на информацию пользовательского профиля, хранящуюся в реестре, поэтому, если профиль еще не загружен, следующий шаг Lsasrv — загрузка в реестр профиля олицетворяемого пользователя вызовом функции `LoadUserProfiIe` из Userenv.dll. Обычно профиль пользователя к этому моменту уже загружен, поскольку Winlogon загружает его при входе пользователя в систему. Но если пользователь регистрируется под другой учетной записью с помощью команды RunAs, то при попытке обращения к зашифрованному файлу под этой учетной записью соответствующий профиль может быть не загружен.

После этого Lsasrv генерирует для файла FEK, обращаясь к средствам шифрования RSA, реализованным в Microsoft Base Cryptographic Provider 1.0.

Создание связок ключей

К этому моменту Lsasrv уже получил FEK и может сгенерировать информацию EFS, сохраняемую вместе с файлом, включая зашифрованную версию FEK. Lsasrv считывает из параметра реестра `HKCU\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash` значение, присвоенное пользователю, который затребовал операцию шифрования, и получает сигнатуру открытого ключа этого пользователя. Lsasrv использует эту сигнатуру для доступа к открытому ключу пользователя и для шифрования FEK.

Теперь Lsasrv может создать информацию, которую EFS сохранит вместе с файлом. EFS хранит в шифрованном файле только один блок информации, в котором содержатся записи для всех пользователей этого файла. Данные записи называются **элементами ключей** (key entries); они хранятся в области сопоставленных с файлом данных EFS, которая называется **Data Decryption Field (DDF).** Совокупность нескольких элементов ключей называется **связкой ключей** (key ring), поскольку, как уже говорилось, EFS позволяет нескольким лицам совместно использовать шифрованный файл.

Формат данных EFS, сопоставленных с файлом, и формат элемента ключа показан на Рис. 31. В первой части элемента ключа EFS хранит информацию, достаточную для точного описания открытого ключа пользователя. В нее входит SID пользователя (его наличие не гарантируется), имя контейнера, в котором хранится ключ, имя провайдера криптографических сервисов и хэш сертификата криптографической пары (при расшифровке используется только этот хэш). Во второй части элемента ключа содержится шифрованная версия FEK. Lsasrv шифрует FEK через CryptoAPI по алгоритму RSA с применением открытого ключа данного пользователя.

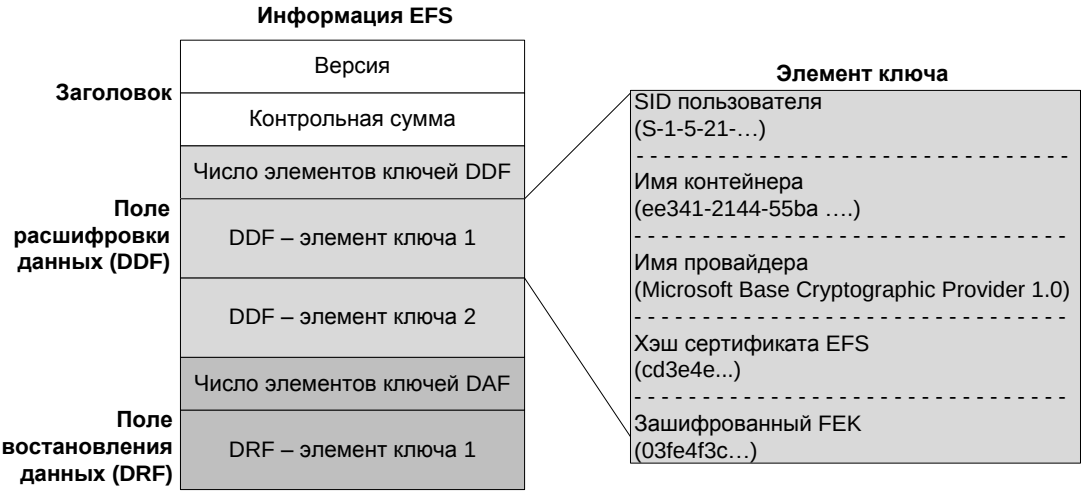


Рис. 31. Формат информации EFS и элемента ключа

Далее Lsasrv создает еще одну связку ключей, содержащую элементы ключей восстановления (recovery key entries). EFS хранит информацию об этих элементах в поле DRF файла (см. Рис. 31). Формат элементов DRF идентичен формату DDE DRF служит для расшифровки пользовательских данных по определенным учетным записям (агентов восстановления) в тех случаях, когда администратору нужен доступ к пользовательским данным. Допустим, сотрудник компании забыл свой пароль для входа в систему. В этом случае администратор может сбросить пароль этого сотрудника, но без агентов восстановления никто не сумеет восстановить его зашифрованные данные.

Шифрование файловых данных

Ход процесса шифрования показан на Рис. 32. После создания всех данных, необходимых для шифруемого пользователем файла, Lsasrv приступает к шифрованию файла и создает его резервную копию, Efs0.tmp. Резервная копия помещается в тот каталог, где находится шифруемый файл. Lsasrv применяет к резервной копии ограничивающий дескриптор защиты, так что доступ к этому файлу можно получить только по учетной записи System. Далее Lsasrv инициализирует файл журнала, созданный им на первом этапе процесса шифрования и регистрирует в нем факт создания резервного файла. Lsasrv шифрует исходный файл только после его резервирования.

Lsasrv посылает через NTFS коду EFS режима ядра команду на добавление к исходному файлу созданной информации EFS. В Windows 2000 NTFS получает эту команду, но поскольку она не понимает команд EFS, то просто вызывает драйвер EFS. Код EFS режима ядра принимает посланные Lsasrv данные EFS и применяет их к файлу через функции, экспортируемые NTFS. Эти функции позволяют EFS добавлять к NTFS-файлам атрибут $EFS. После этого управление возвращается к Lsasrv, который копирует содержимое шифруемого файла в резервный. Закончив создание резервной копии (в том числе скопировав все дополнительные потоки данных), Lsasrv отмечает в файле журнала, что резервный файл находится в актуальном состоянии. Затем Lsasrv посылает NTFS другую команду, требуя зашифровать содержимое исходного файла.

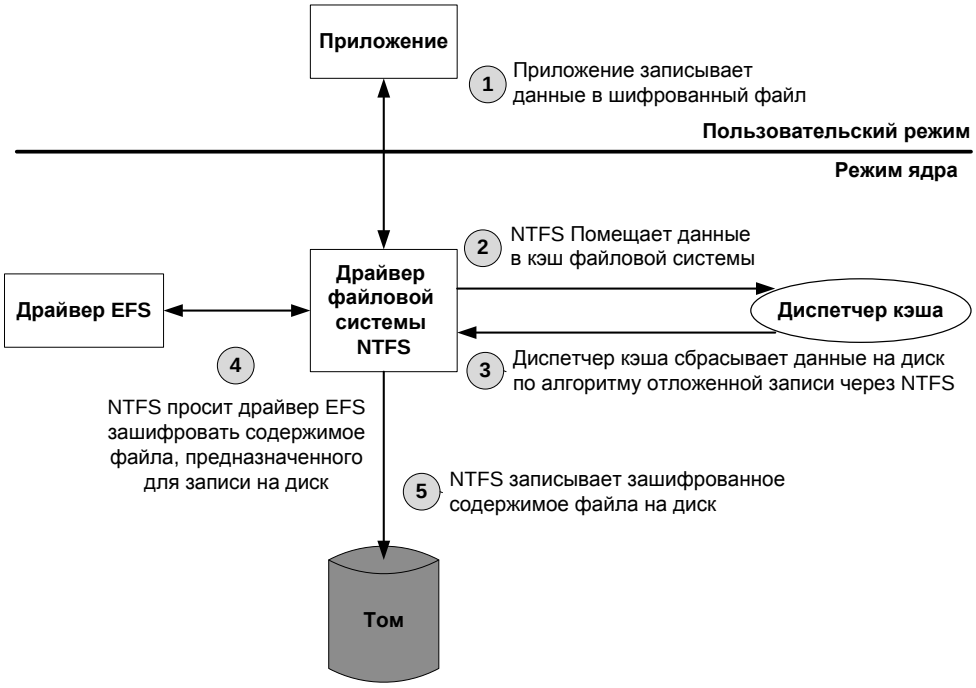


Рис. 32. Схема работы EFS

Получив от EFS команду на шифрование файла, NTFS удаляет содержимое исходного файла и копирует в него данные резервного. По мере копирования каждого раздела файла NTFS сбрасывает данные раздела из кэша файловой системы, и они записываются на диск. Поскольку файл помечен как шифрованный, NTFS вызывает EFS для шифрования раздела данных перед записью на диск. EFS использует незашифрованный FEK, переданный NTFS, чтобы шифровать файловые данные по алгоритму DESX, AES или 3DES порциями, равными по размеру одному сектору (512 байтов).

После того как файл зашифрован, Lsasrv регистрирует в файле журнала, что шифрование успешно завершено, и удаляет резервную копию файла. В заключение Lsasrv удаляет файл журнала и возвращает управление приложению, запросившему шифрование файла.

Сводная схема процесса шифрования

Рассмотрим сводный список этапов шифрования файла через EFS.

1. Загружается профиль пользователя, если это необходимо.

2. В каталоге System Volume Information создается файл журнала с именем Efsxlog, где `x`— уникальное целое число от 0. По мере выполнения следующих этапов в журнал заносятся записи, позволяющие восстановить файл после сбоя системы в процессе шифрования.

3. Base Cryptographic Provider 1.0 генерирует для файла случайное 128-битное число, используемое в качестве FEK.

4. Генерируется или считывается криптографическая пара ключей пользователя. Она идентифицируется в `HKCU\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash`.

5. Для файла создается связка ключей DDF с элементом для данного пользователя. Этот элемент содержит копию FEK, зашифрованную с помощью открытого EFS-ключа пользователя.

6. Для файла создается связка ключей DRF. В нем есть элементы для каждого агента восстановления в системе, и при этом в каждом элементе содержится копия FEK, зашифрованная с помощью открытого EFS-ключа агента.

7. Создается резервный файл с именем вида Efs0.tmp в том каталоге, где находится и шифруемый файл.

8. Связки ключей DDF и DRF добавляются к заголовку и сопоставляются с файлом как атрибут EFS.

9. Резервный файл помечается как шифрованный, и в него копируется содержимое исходного файла.

10. Содержимое исходного файла уничтожается, в него копируется содержимое резервного. В результате этой операции данные исходного файла шифруются, так как теперь файл помечен как шифрованный.

11. Удаляется резервный файл.

12. Удаляется файл журнала.

13. Выгружается профиль пользователя (загруженный на этапе 1).

Процесс расшифровки

Процесс расшифровки начинается, когда пользователь открывает шифрованный файл. При открытии файла NTFS анализирует его атрибуты и выполняет функцию обратного вызова в драйвере EFS. Драйвер EFS считывает атрибут $EFS, сопоставленный с шифрованным файлом. Чтобы прочитать этот атрибут, драйвер вызывает функции поддержки EFS, которые NTFS экспортирует для EFS. NTFS выполняет все необходимые действия, чтобы открыть файл. Драйвер EFS проверяет наличие у пользователя, открывающего файл, прав доступа к данным шифрованного файла. После такой проверки EFS получает расшифрованный FEK файла, применяемый для обработки данных в операциях, которые пользователь может выполнять над файлом.

EFS не может расшифровать FEK самостоятельно и полагается в этом на Lsasrv (который может использовать `CryptoAPI`). С помощью драйвера Ksecdd.sys EFS посылает LPC-сообщение Lsasrv, чтобы тот извлек из атрибута IEFS (т. е. из данных EFS) FEK пользователя, открывающего файл, и расшифровал его.

Получив LPC-сообщение, Lsasrv вызывает функцию `LoadUserProfile` из Use-renv.dll для загрузки в реестр профиля пользователя, если он еще не загружен. Lsasrv перебирает все поля ключей в данных EFS, пробуя расшифровать каждый FEK на основе закрытого ключа пользователя; с этой целью Lsasrv пытается расшифровать FEK в DDF- или DRF-элементе ключа. Если хэш сертификата в поле ключа не подходит к ключу пользователя, Lsasrv переходит к следующему полю ключа. Если Lsasrv не удастся расшифровать ни одного FEK в DDF или DRF, пользователь не получит FEK файла, и EFS запретит доступ к файлу приложению, которое пыталось открыть этот файл. А если Lsasrv найдет какой-нибудь хэш, который соответствует ключу пользователя, он расшифрует FEK по закрытому ключу пользователя через `CryptoAPI`.

Lsasrv, обрабатывая при расшифровке FEK связки ключей DDF и DRF, автоматически выполняет операции восстановления файла. Если к файлу пытается получить доступ агент восстановления, не зарегистрированный на доступ к шифрованному файлу (т. е. у него нет соответствующего поля в связке ключей DDF), EFS позволит ему обратиться к файлу, потому что агент имеет доступ к паре ключей для поля ключа в связке ключей DRE.

Расшифровка файловых данных

Открыв шифрованный файл, приложение может читать и записывать его данные. Для расшифровки файловых данных NTFS вызывает драйвер EFS по мере чтения этих данных с диска — до того, как помещает их в кэш файловой системы. Аналогичным образом, когда приложение записывает данные в файл, они остаются незашифрованными в кэше файловой системы, пока приложение или диспетчер кэша не сбросит данные обратно на диск с помощью NTFS. При записи данных

шифрованного файла из кэша на диск NTFS вызывает драйвер EFS, чтобы зашифровать их.

Как уже говорилось, драйвер EFS выполняет шифрование и расшифровку данных порциями по 512 байтов. Такой размер оптимален для драйвера, потому что объем данных при операциях чтения и записи кратен размеру сектора.

Резервное копирование шифрованных файлов

EFS предоставляет утилитам резервного копирования механизм, с помощью которого они могут создавать резервные копии файлов и восстанавливать их в шифрованном виде. Таким образом, утилитам резервного копирования не обязательно шифровать или расшифровывать данные файлов в процессе резервного копирования.

Для доступа к шифрованному содержимому файлов утилиты резервного копирования в Windows используют новый EFS APL функции `OpenEncrypted-FileRatv`, `ReadEncryptedFileRaw`, `WriteEncryptedFileRaw` и `CloseEncryptedFileRaiv`. Эти функции, предоставляемые Advapi32.dll, вызывают соответствующие функции Lsasrv по механизму LPC

`EfsReadFileRaw` может понадобиться несколько операций чтения, чтобы считать большой файл. По мере того как `EfsReadFileRaw` считывает очередную порцию файла, Lsasrv посылает Advapi32.dll RPC-сообщение, в результате которого выполняется функция обратного вызова, указанная программой резервного копирования при вызове `ReadEncryptedFileRaw`. Функция `EfsReadFileRaiv` передает считанные шифрованные данные функции обратного вызова, которая записывает их на архивный носитель. Восстанавливаются шифрованные файлы аналогичным образом. Программа резервного копирования вызывает API-функцию `WriteEncryptedFileRaw`, которая активизирует функцию обратного вызова программы резервного копирования для получения нешифрованных данных с архивного носителя, в то время как Lsasrv-функция `EfsWriteFileRaw` восстанавливает содержимое файла.

Литература

1. Э. Таненбаум. Современные операционные системы. 3-ое изд. –СПб.: Питер, 2010. – 1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. М. Руссинович, Д. Соломон. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP, Windows 2000. Мастер-класс. / Пер. с англ. -4-е изд. –М.: Издательско-торговый дом «Русская редакция»; СПб.: Питер; 2005. -992 с.

4. Microsoft Development Network. URL: http://msdn.com

5. NCSC. URL: http://www.radium.ncsc.mil/tpep

6. Рейтинги степени защищенности коммерческих ОС, сетевых компонентов и приложений. URL: http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html

7. Common Criteria. URL: http//csrc.nistgov/cc

8. Результаты оценки Windows 2000 на соответствие требованиям Controlled Access PP. URL: http://niap.nist.gov/cc-scheme.html

9. Результаты оценки Windows 2000 на соответствие дополнительным требованиям CC: http://niap.nist.gov/cc-scheme/CCEVS-VID402-ST.pdf

10. Результаты оценки Windows XP & Windows Server 2003. URL: http://niap.nist gov/cc-scheme/in_evaluation.html

11. RFC1510. URL: http://www.ietf.org