Глава 10: Введение в файловую систему и мини-фильтры.

Файловые системы предоставляют операции ввода-вывода для доступа к файлам. Windows поддерживает несколько файловых систем, прежде всего это NTFS.

Фильтрация файловой системы - это механизм, с помощью которого драйверы могут перехватывать вызовы, адресованные файловой системе. Это полезно для многих типов программного обеспечения, таких как антивирусы, резервное копирование, шифрование и многое другое.

Windows долгое время поддерживала модель фильтрации, известную как фильтры файловой системы, которая сейчас называется устаревшими фильтрами файловой системы.

Была разработана новая модель, называемая мини-фильтрами файловой системы для замены устаревшего механизма фильтрации. Мини-фильтры легче писать во многих отношениях, и они предпочтительный способ разработки драйверов фильтрации файловой системы.

В этой главе:

• Введение.

• Загрузка и выгрузка.

• Инициализация.

• Установка.

• Обработка операций ввода-вывода.

• Драйвер защиты от удаления.

• Имена файлов.

• Контексты.

• Инициирование запросов ввода-вывода.

• Драйвер резервного копирования файлов.

• Связь в пользовательском режиме.

• Отладка.

• Упражнения.

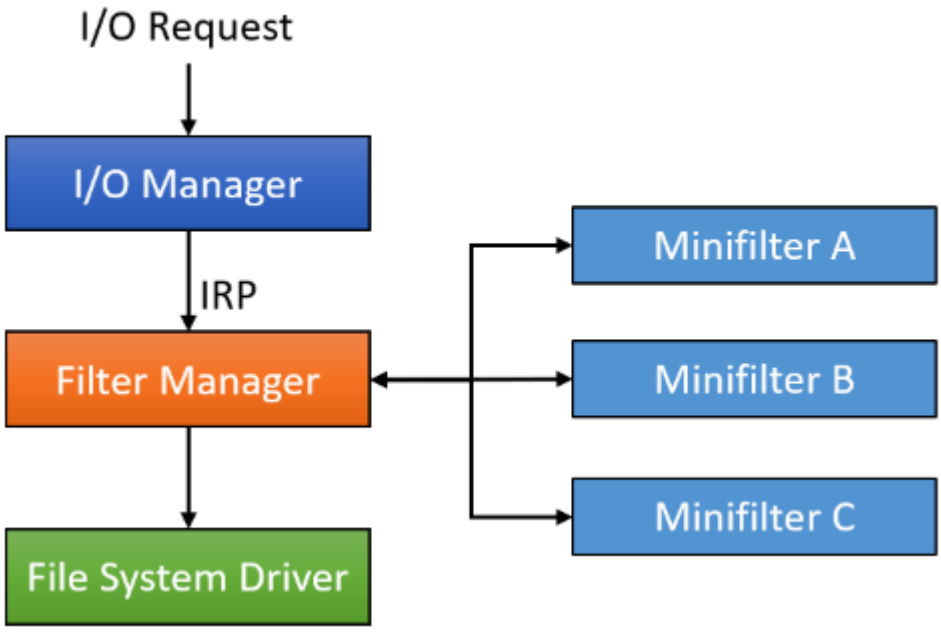
Введение

Известно, что фильтры устаревшей файловой системы сложно написать.

Разработчик драйвера должен позаботиться о наборе мелких деталей, многие из которых являются шаблонными, что усложняет разработку. Устаревшие фильтры не могут быть выгружены во время работы системы, что означает, что систему пришлось перезапустить, чтобы загрузить обновленную версию драйвера.

Модель с мини-фильтром позволяет загружать и выгружать драйверы динамически, что значительно упрощает рабочий процесс разработки.

Внутренне устаревший фильтр, предоставляемый Windows, называемый диспетчером фильтров, отвечает за управление мини-фильтров. Типичное наслоение фильтров показано на следующем рисунке.



Каждый мини-фильтр имеет собственную высоту, которая определяет его относительное положение в стеке устройства.

Диспетчер фильтров - это тот, кто получает пакеты IRP, как и любой другой устаревший фильтр, а затем вызывает мини-фильтры, которыми он управляет, в порядке убывания высоты. В некоторых необычных случаях в иерархии может быть другой устаревший фильтр, который может вызвать мини- фильтр «сплит», где некоторые из них выше по высоте, чем унаследованные фильтры, а некоторые ниже. В таком случае будет загружено более одного экземпляра диспетчера фильтров, каждый из которых управляет своими мини-фильтрами. Каждый такой экземпляр диспетчера фильтров называется фреймом.

На следующем рисунке показан такой пример с двумя фреймами.

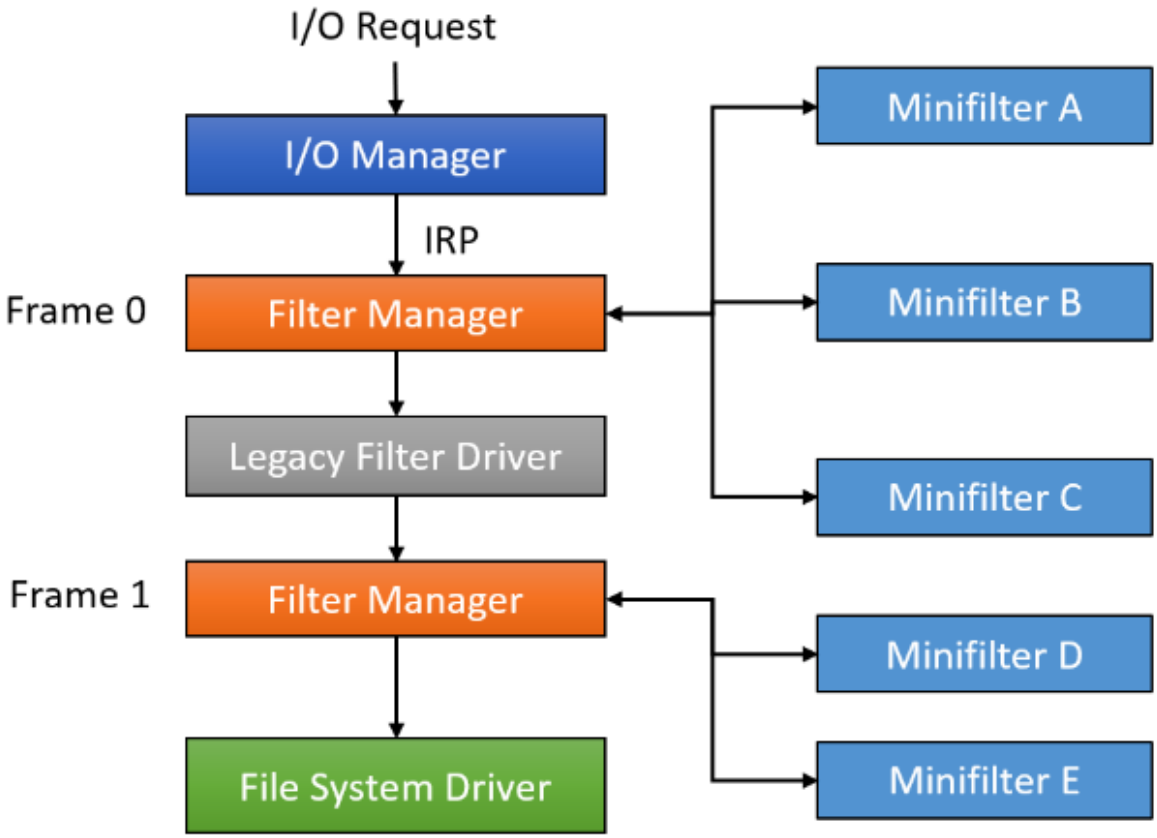


Figure 10-2: Mini-filters in two filter manager frames

Загрузка и выгрузка

Драйверы мини-фильтра должны быть загружены так же, как и любой другой драйвер. Для этого используется API пользовательского режима - FilterLoad, куда нужно передать имя драйвера (его ключ в реестре в HKLM\System\CurrentControlSet\Services\Drivername).

Внутри вызывается API ядра FltLoadFilter с той же семантикой. Как и любой другой драйвер, SeLoadDriverPrivilege должен присутствовать в токене вызывающего, если вызывается из пользовательского режима. По умолчанию он присутствует в токенах уровня администратора, но не в токенах обычных пользователей.

Выгрузка мини-фильтра осуществляется с помощью FilterUnload в пользовательском режиме или FltUnload в режиме ядра. Эта операция требует тех же привилегий, что и для загрузок, но не гарантируется, что будет успех, потому что вызывается обратный вызов выгрузки фильтра мини-фильтра (Будет обсуждаться позже).

Хотя использование API для загрузки и выгрузки фильтров имеет свое применение, во время разработки обычно проще используйте встроенный инструмент, который может сделать это (и многое другое) под названием fltmc.exe.

Если вызвать его (из-под администратора) без аргументов, то он перечисляет загруженные в данный момент мини-фильтры. Вот результат на моём компьютер с Windows 10 Pro версии 1903:

```
C:\WINDOWS\system32>fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                             1       409800         0
FsDepends                           9       407000         0
WdFilter                           10       328010         0
storqosflt                          1       244000         0
wcifs                               3       189900         0
PrjFlt                              1       189800         0
CldFlt                              2       180451         0
FileCrypt                           0       141100         0
luafv                               1       135000         0
npsvctrig                           1        46000         0
Wof                                 8        40700         0
FileInfo                           10        40500         0
```

Для каждого фильтра в выходных данных отображается имя драйвера, количество экземпляров каждого фильтра в настоящее время работы, его высоту и кадр диспетчера фильтров, частью которого он является.

Вам может быть интересно, почему существуют драйверы с разным количеством экземпляров.

Загрузка драйвера с помощью fltmc.exe выполняется с параметром загрузки, например:

fltmc load myfilter

И наоборот, выгрузка выполняется с помощью параметра командной строки unload:

fltmc unload myfilter

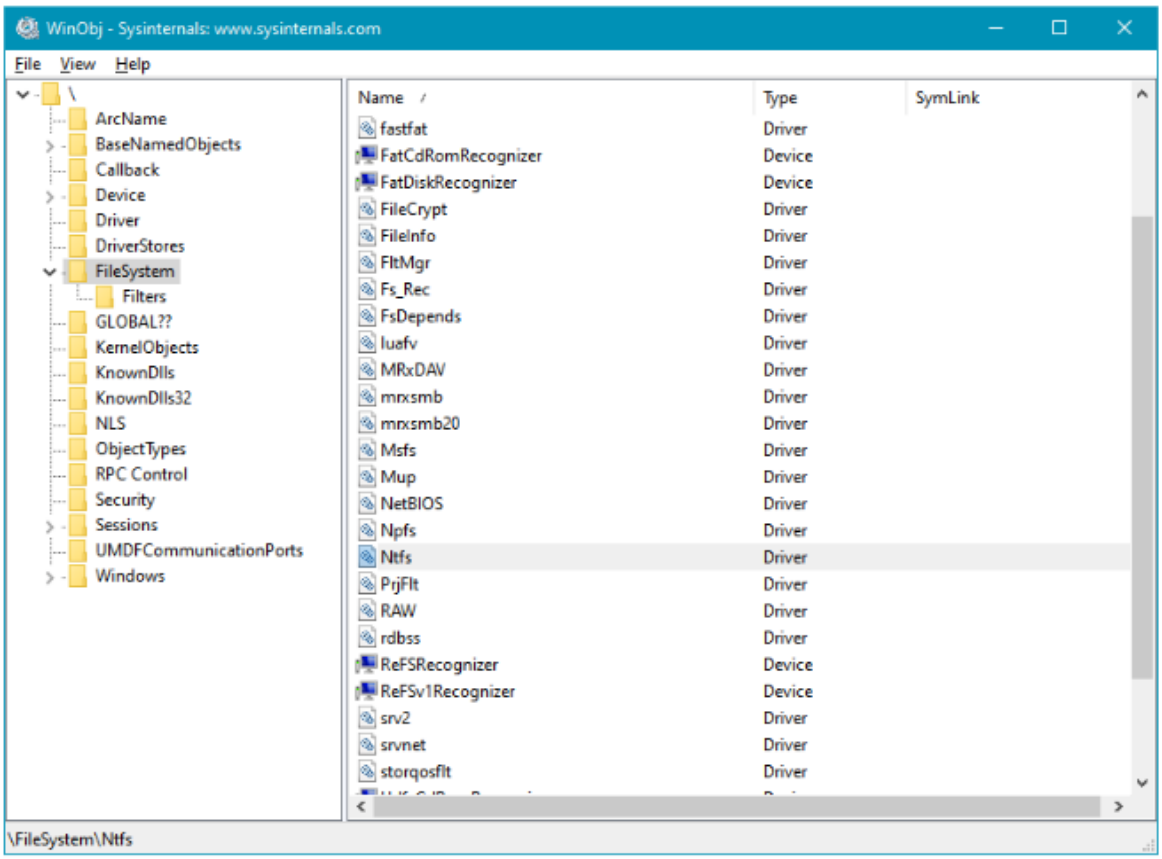
fltmc включает и другие параметры. Введите fltmc -? чтобы получить полный список параметров.

Например, вы можете получить подробную информацию всех экземпляров для каждого драйвера, использующего экземпляры fltmc.

Аналогичным образом вы можете получить список всех томов установленных в системе.

Драйверы и фильтры файловой системы создаются в каталоге FileSystem диспетчера объектов.

На следующем рисунке показан этот каталог в WinObj.



Инициализация

Драйвер мини-фильтра файловой системы имеет подпрограмму DriverEntry, как и любой другой драйвер. Драйвер должен зарегистрироваться как мини-фильтр в диспетчере фильтров, задав различные настройки, такие как, какие операции, которые он хочет перехватить.

Драйвер устанавливает соответствующие структуры, а затем вызывает FltRegisterFilter для регистрации. В случае успеха драйвер может выполнять дальнейшие инициализации по мере необходимости, и, наконец, вызвать FltStartFiltering, чтобы фактически начать операции фильтрации.

Обратите внимание, что драйверу нет необходимости устанавливать собственные процедуры отправки (IRP_MJ_READ, IRP_MJ_WRITE и т. д.).

Функция FltRegisterFilter имеет следующий прототип:

```
NTSTATUS FltRegisterFilter (
            _In_ PDRIVER_OBJECT Driver,
            _In_ const FLT_REGISTRATION *Registration,
            _Outptr_ PFLT_FILTER *RetFilte);
```

Требуемая структура FLT_REGISTRATION предоставляет всю необходимую информацию для регистрации. Она определяется так:

```
typedef struct _FLT_REGISTRATION {

	USHORT Size;

	USHORT Version;

	FLT_REGISTRATION_FLAGS Flags;

	const FLT_CONTEXT_REGISTRATION *ContextRegistration;

	const FLT_OPERATION_REGISTRATION *OperationRegistration;

	PFLT_FILTER_UNLOAD_CALLBACK FilterUnloadCallback;

	PFLT_INSTANCE_SETUP_CALLBACK InstanceSetupCallback;

	PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK InstanceQueryTeardownCallback;

	PFLT_INSTANCE_TEARDOWN_CALLBACK InstanceTeardownStartCallback;

	PFLT_INSTANCE_TEARDOWN_CALLBACK InstanceTeardownCompleteCallback;

	PFLT_GENERATE_FILE_NAME GenerateFileNameCallback;

	PFLT_NORMALIZE_NAME_COMPONENT NormalizeNameComponentCallback;

	PFLT_NORMALIZE_CONTEXT_CLEANUP NormalizeContextCleanupCallback;

	PFLT_TRANSACTION_NOTIFICATION_CALLBACK TransactionNotificationCallback;

	PFLT_NORMALIZE_NAME_COMPONENT_EX NormalizeNameComponentExCallback;

#if FLT_MGR_WIN8

	PFLT_SECTION_CONFLICT_NOTIFICATION_CALLBACK
SectionNotificationCallback;

#endif

} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

В этой структуре заключено много информации.

Важные поля описаны ниже:

• Размер должен соответствовать размеру структуры, который может зависеть от целевой версии Windows.Обычно драйверы просто указывают sizeof (FLT_REGISTRATION).

• Версия также зависит от целевой версии Windows. Драйверы используют FLT_REGISTRATION_VERSION.

• Флаги могут быть нулевыми или представлять собой комбинацию следующих значений:

- FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP - драйвер не поддерживает остановку запроса, независимо от других настроек.

- FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS - драйвер знает именованные каналы хочет также фильтровать запросы к этим файловым системам.

- FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME (Windows 10 версии 1607 и более поздних) - драйвер будет поддерживать подключение к тому прямого доступа (DAX), если такой том доступен.

• ContextRegistration - необязательный указатель на массив структур FLT_CONTEXT_REGISTRATION, где каждая запись представляет собой контекст, который драйвер может использовать в своей работе. Контекст относится к некоторым определяемым драйвером данных, которые могут быть прикреплены к объектам файловой системы, таким как файлы и тома. Некоторым драйверам не нужны контексты, и они могут установить это значение NULL.

• OperationRegistration - безусловно, самая важная область. Это указатель на массив структуры FLT_OPERATION_REGISTRATION, каждая из которых определяет интересующую операцию.

• FilterUnloadCallback - указывает функцию, которая будет вызываться, когда драйвер собирается быть выгружен. Если указано NULL, драйвер не может быть выгружен. Если драйвер устанавливает обратный вызов и возвращает успешный статус, то драйвер будет выгружен успешно, в этом случае он должен выполнить FltUnregisterFilter, чтобы отменить регистрацию перед выгрузкой. Возвращение ошибки в status не выгружает драйвер.

• InstanceSetupCallback - этот обратный вызов позволяет драйверу получать уведомления, когда экземпляр будет прикреплен к новому тому. Драйвер может вернуть STATUS_SUCCESS для подключения или STATUSFLT_DO_NOT_ATTACH, если драйвер не желает подключаться к этому тому.

• InstanceQueryTeardownCallback - дополнительный обратный вызов, вызываемый перед отсоединением от тома. Это может произойти из-за явного запроса на отключение с помощью FltDetachVolume в режиме ядра или FilterDetach в пользовательском режиме. Если обратным вызовом указано значение NULL, операция отсоединение будет прервана.

• InstanceTeardownStartCallback - дополнительный обратный вызов, вызываемый при начале отсоединения экземпляра. Драйвер должен завершить все отложенные операции, чтобы отсоединение экземпляра могло-быть полным. Указание NULL для этого обратного вызова не предотвращает отсоединение экземпляра.

• InstanceTeardownCompleteCallback - дополнительный обратный вызов, вызываемый после всех ожидающих операций ввода-вывода.

Остальные поля обратного вызова необязательны и используются редко. Это выходит за рамки этой книги.

Регистрация обратного вызова

Драйвер мини-фильтра должен указывать, какие операции ему интересны. Это предоставляется в мини-фильтре.

```
typedef struct _FLT_OPERATION_REGISTRATION {
      UCHAR MajorFunction;
      FLT_OPERATION_REGISTRATION_FLAGS Flags;
      PFLT_PRE_OPERATION_CALLBACK PreOperation;
      PFLT_POST_OPERATION_CALLBACK PostOperation;
      // reserved
      PVOID Reserved1;
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

Сама операция идентифицируется основным кодом функции, многие из которых такие же, как и мы встречались в предыдущих главах: IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE и так далее.

Тем не менее, есть другие операции, отождествляемые с основной функцией, которые не имеют реальной основной функции. Эта абстракция, предоставляемая диспетчером фильтров, помогает изолировать мини-фильтр.

Кроме того, файловые системы поддерживают еще один механизм для получение запросов, известное как Fast I/O.

Быстрый ввод-вывод используется для синхронного ввода-вывода с кэшированными файлами. Запросы ввода-вывода передают данные между пользовательскими буферами и системным кешем напрямую, минуя файловый стек драйверов и хранилища, что позволяет избежать ненужных накладных расходов.

Драйвер файловой системы NTFS, в качестве примера поддерживает Fast I/O.

Эту информацию можно просмотреть с помощью отладчика ядра с помощью команды ! drvobj, как показано здесь для драйвера файловой системы NTFS:

```
lkd> !drvobj \filesystem\ntfs f
Driver object (ffffad8b19a60bb0) is for:
 \FileSystem\Ntfs

Driver Extension List: (id , addr)

Device Object list:
ffffad8c22448050  ffffad8c476e3050  ffffad8c3943f050  ffffad8c208f1050
ffffad8b39e03050  ffffad8b39e87050  ffffad8b39e73050  ffffad8b39d52050
ffffad8b19fc9050  ffffad8b199f3d80

DriverEntry:   fffff8026b609010 Ntfs!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                fffff8026b49bae0  Ntfs!NtfsFsdCreate
[01] IRP_MJ_CREATE_NAMED_PIPE     fffff80269141d40  nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                 fffff8026b49d730  Ntfs!NtfsFsdClose
[03] IRP_MJ_READ                  fffff8026b3b3f80  Ntfs!NtfsFsdRead

(truncated)

[19] IRP_MJ_QUERY_QUOTA           fffff8026b49c700  Ntfs!NtfsFsdDispatchWait
[1a] IRP_MJ_SET_QUOTA             fffff8026b49c700  Ntfs!NtfsFsdDispatchWait
```

```
[1b] IRP_MJ_PNP                 fffff8026b5143e0  Ntfs!NtfsFsdPnp

Fast I/O routines:
FastIoCheckIfPossible           fffff8026b5adff0  Ntfs!NtfsFastIoCheckIfPossible
FastIoRead                      fffff8026b49e080  Ntfs!NtfsCopyReadA
FastIoWrite                     fffff8026b46cb00  Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo            fffff8026b4d50d0  Ntfs!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo         fffff8026b4d2de0  Ntfs!NtfsFastQueryStdInfo
FastIoLock                      fffff8026b4d6160  Ntfs!NtfsFastLock
FastIoUnlockSingle              fffff8026b4d6b40  Ntfs!NtfsFastUnlockSingle
FastIoUnlockAll                 fffff8026b5ad2d0  Ntfs!NtfsFastUnlockAll
FastIoUnlockAllByKey            fffff8026b5ad590  Ntfs!NtfsFastUnlockAllByKey
ReleaseFileForNtCreateSection   fffff8026b3c3670  Ntfs!NtfsReleaseForCreateSection
FastIoQueryNetworkOpenInfo      fffff8026b4d4cb0  Ntfs!NtfsFastQueryNetworkOpenInfo
AcquireForModWrite              fffff8026b3c4c20  Ntfs!NtfsAcquireFileForModWrite
MdlRead                         fffff8026b46b6a0  Ntfs!NtfsMdlReadA
MdlReadComplete                 fffff8026911aca0  nt!FsRtlMdlReadCompleteDev
PrepareMdlWrite                 fffff8026b46aae0  Ntfs!NtfsPrepareMdlWriteA
MdlWriteComplete                fffff802696c41e0  nt!FsRtlMdlWriteCompleteDev
FastIoQueryOpen                 fffff8026b4d4940  Ntfs!NtfsNetworkOpenCreate
ReleaseForModWrite              fffff8026b3c5a40  Ntfs!NtfsReleaseFileForModWrite
AcquireForCcFlush               fffff8026b3a8690  Ntfs!NtfsAcquireFileForCcFlush
ReleaseForCcFlush               fffff8026b3c5610  Ntfs!NtfsReleaseFileForCcFlush

Device Object stacks:

!devstack ffffad8c22448050 :
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffad8c4adcba70  \FileSystem\FltMgr ffffad8c4adcbbc0
> ffffad8c22448050  \FileSystem\Ntfs   ffffad8c224481a0

(truncated)

Processed 10 device objects.
```

Диспетчер фильтров абстрагирует операции ввода-вывода, независимо от того, основаны ли они на IRP или на быстром вводе-выводе. Мини-фильтры могут перехватить любой такой запрос. Если драйвер не заинтересован в быстром вводе-выводе, он может запросить фактический тип запроса, предоставленный диспетчером фильтров, с помощью FLT_IS_FASTIO_OPERATION и/или FLT_IS_IRP_OPERATION.

В таблице перечислены некоторые из основных функций мини-фильтров файловой системы с кратким описанием.

Major function	Dispatch routine?	Description
IRP_MJ_CREATE	Yes	Create or open a file/directory
IRP_MJ_READ	Yes	Read from a file
IRP_MJ_WRITE	Yes	Write to a file
IRP_MJ_QUERY_EA	Yes	Read extended attributes from a file/directory
IRP_MJ_DIRECTORY_CONTROL	Yes	Request sent to a directory
IRP_MJ_FILE_SYSTEM_CONTROL	Yes	File system device I/O control request
IRP_MJ_SET_INFORMATION	Yes	Various information setting for a file (e.g. delete, rename)
IRP_MJ_ACQUIRE_FOR_SECTION_-SYNCHRONIZATION	No	Section (memory mapped file) is being opened
IRP_MJ_OPERATION_END	No	signals the end of array of operations callbacks

Второе поле в FLT_OPERATION_REGISTRATION - это набор флагов, которые могут быть нулем или комбинация одного из следующих флагов, влияющих на операции чтения и записи:

• FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO - не вызывать обратные вызовы, если они являются кэшированным I/O (например, быстрые операции ввода-вывода, которые всегда кэшируются).

• FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO - не вызывать обратный вызов (ы) для страничного ввода/вывода (только операции на основе IRP).

• FLTFL_OPERATION_REGISTRATION_SKIP_NON_DASD_IO - не вызывать обратный вызов (ы) для DAX томов. Следующие два поля - это обратные вызовы до и после операции, где хотя бы одно должно быть отличным от NULL (иначе зачем вообще эта запись?).

Вот пример инициализации массива структуры FLT_OPERATION_REGISTRATION (для воображаемого драйвера под названием «Sample»):

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {

            { IRP_MJ_CREATE, 0, nullptr, SamplePostCreateOperation },

            { IRP_MJ_WRITE, FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO,

                        SamplePreWriteOperation, nullptr },

                        { IRP_MJ_CLOSE, 0, nullptr,
SamplePostCloseOperation },

                        { IRP_MJ_OPERATION_END }

};
```

Имея под рукой этот массив, можно зарегистрировать драйвер, не требующий каких-либо контекстов со следующим кодом:

```
const FLT_REGISTRATION FilterRegistration = {

            sizeof(FLT_REGISTRATION),

            FLT_REGISTRATION_VERSION,

            0,

            // Flags

            nullptr,

            // Context

            Callbacks,

            // Operation callbacks

            ProtectorUnload,

            // MiniFilterUnload

            SampleInstanceSetup,

            // InstanceSetup

            SampleInstanceQueryTeardown,

            // InstanceQueryTeardown

            SampleInstanceTeardownStart,

            // InstanceTeardownStart

            SampleInstanceTeardownComplete, // InstanceTeardownComplete

};

PFLT_FILTER FilterHandle;

NTSTATUS

DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING
RegistryPath) {

      NTSTATUS status;

      //... some code

      status = FltRegisterFilter(DriverObject, &FilterRegistration,
&FilterHandle);

      if(NT_SUCCESS(status)) {

            // actually start I/O filtering

            status = FltStartFiltering(FilterHandle);
```

```
            if(!NT_SUCCESS(status))
                  FltUnregisterFilter(FilterHandle);
      }
      return status;
}
```

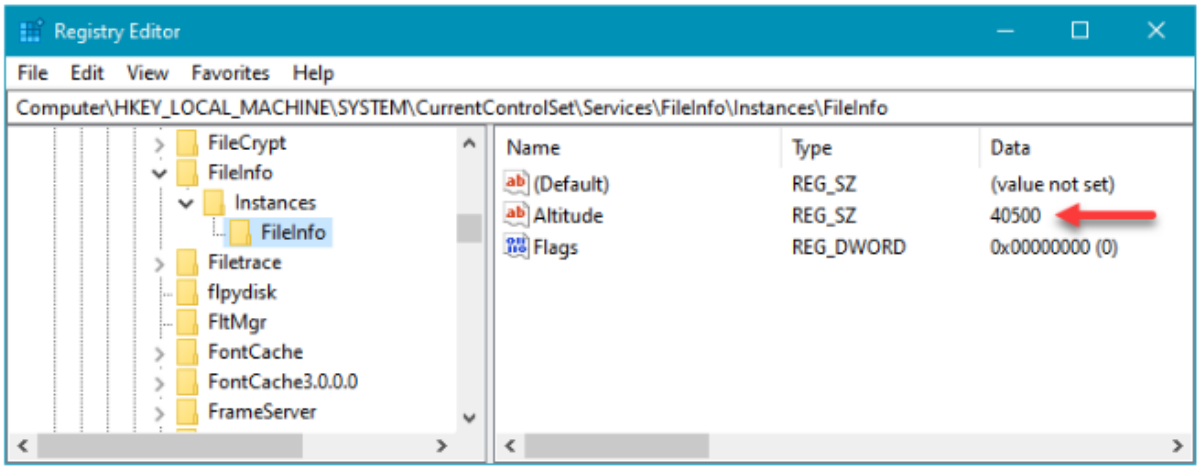
Высота

Как мы уже видели, мини-фильтры файловой системы должны иметь высоту, указывающую их относительное «положение» в иерархии фильтров файловой системы.

В отличие от высоты, с которой мы уже столкнулись с обратными вызовами объекта и реестра, значение высоты мини-фильтра может быть потенциально значительным.

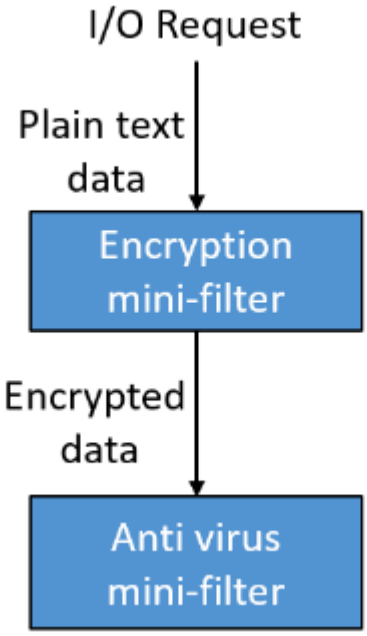
Во-первых, значение высоты не предоставляется при регистрации мини-фильтра, а считывается из реестра.

Когда драйвер установлен, его высота записывается в соответствующем месте в реестре. На рисунке показана запись в реестре для встроенного драйвера мини-фильтра Fileinfo. Высота имеет то же значение, которое было показано ранее для инструмента fltmc.exe.



Вот пример, который должен прояснить, почему высота имеет значение. Допустим, есть мини-фильтр на высоте 10000, чьей задачей является шифрование данных при записи и дешифрование при чтении.

Теперь предположим еще один мини - фильтр, задачей которого является проверка данных на предмет вредоносной активности, находится на высоте 9000. Этот макет изображен на следующем рисунке.



Драйвер шифрования шифрует входящие данные для записи, которые затем передаются антивирусной программе.

Проблема с драйвером антивируса, так как он не видит зашифрованные данные.

В таком случае антивирусный драйвер должен иметь высоту выше, чем драйвер шифрования. Как такой драйвер может гарантировать, что это на самом деле так ?

Чтобы исправить эту (и другие подобные) ситуацию, Microsoft определила диапазоны высот для драйверов исходя из их требований (и, в конечном итоге, их задачи).

Чтобы получить нужную высоту, издатель драйвера должен отправить электронное письмо в Microsoft (fsfcomm@microsoft.com) и запросить высоту, она выделяется для этого драйвера на основе его предполагаемой цели.

Вот ссылка:https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes

Там можно посмотреть драйверы и их высоту, которую выделила Microsoft.

Следующая таблица показывает список групп и диапазон высот для каждой группы.

Altitude range	Group name
420000 - 429999	Filter
400000 - 409999	FSFilter Top
360000 - 389999	FSFilter Activity Monitor
340000 - 349999	FSFilter Undelete
320000 - 329998	FSFilter Anti-Virus
300000 - 309998	FSFilter Replication
280000 - 289998	FSFilter Continuous Backup
260000 - 269998	FSFilter Content Screener
240000 - 249999	FSFilter Quota Management
220000 - 229999	FSFilter System Recovery
200000 - 209999	FSFilter Cluster File System
180000 - 189999	FSFilter HSM
170000 - 174999	FSFilter Imaging (ex: .ZIP)
160000 - 169999	FSFilter Compression
140000 - 149999	FSFilter Encryption
130000 - 139999	FSFilter Virtualization
120000 - 129999	FSFilter Physical Quota management
100000 - 109999	FSFilter Open File
80000 - 89999	FSFilter Security Enhancer
60000 - 69999	FSFilter Copy Protection
40000 - 49999	FSFilter Bottom
20000 - 29999	FSFilter System

Установка

«Правильный» способ установки мини-фильтра файловой системы - использовать файл INF.

Т.к. для его работоспособности нужны записи в реестре.

Файлы INF - это классический механизм, используемый для установки драйверов.

Шаблоны проектов «Мини-фильтр файловой системы», предоставленные WDK создает такой INF-файл, который почти готов к установке.

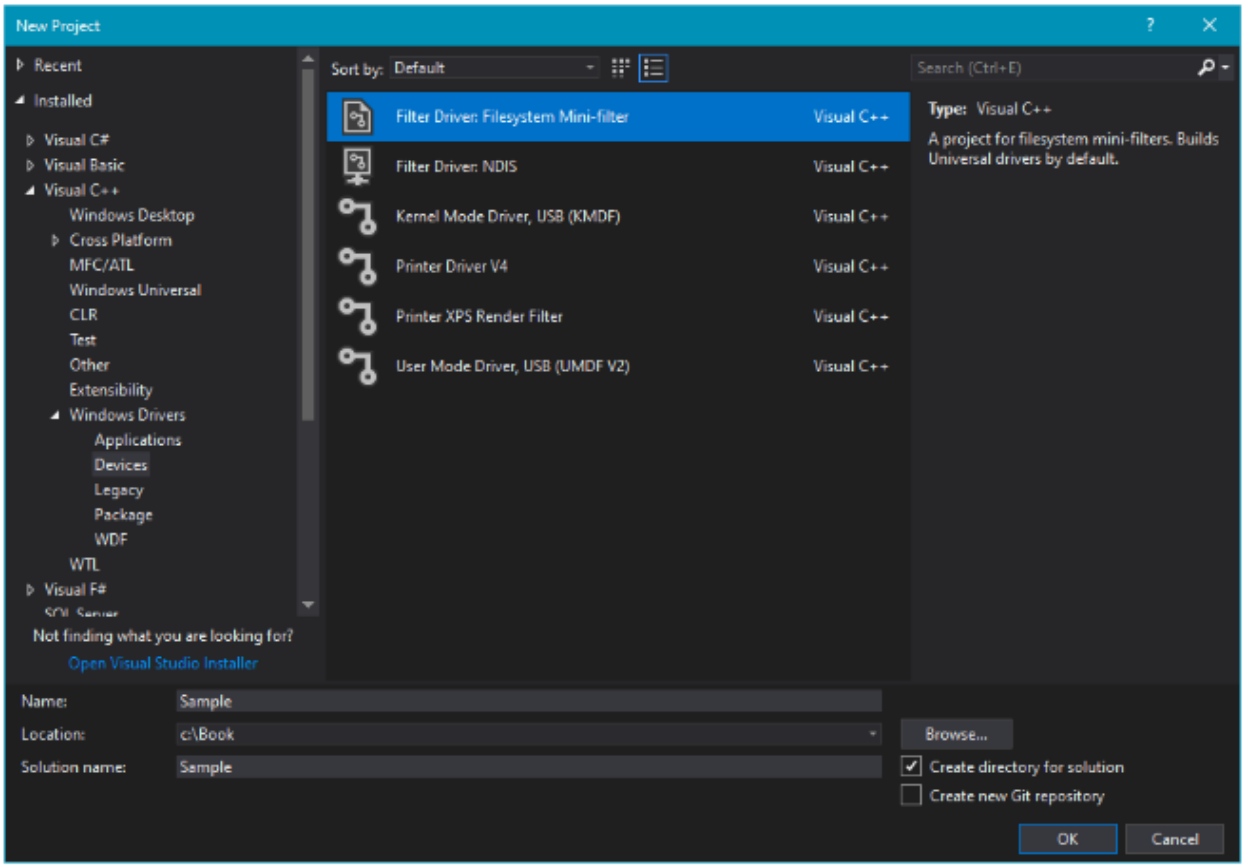
Файлы INF используют старый синтаксис файла INI.

Эти записи представляют собой инструкции для установщика, который анализирует файл, по существу инструктируя установщику выполнить два типа операций: скопировать файлы в определенные местоположения и внесение изменений в реестр.

Давайте рассмотрим INF-файл мини-фильтра файловой системы, созданный мастером проектов WDK.

В Visual Studio найдите узел Драйверы устройств и под ним найдите Устройства. Используемый шаблон - Фильтр. Драйвер: Мини-фильтр файловой системы.

На следующем рисунке показано это в Visual Studio 2017.



Введите имя проекта (образец на рисунке) и нажмите OK. В узле Файлы драйвера в обозревателе решений вы найдете файл INF с именем Sample.inf.

Раздел версии

Раздел версия является обязательным в INF. Следующее создается мастером проекта WDK (немного изменено для удобства чтения):

```
[Version]
Signature
= "$Windows NT$"
; TODO - Change the Class and ClassGuid to match the Load Order Group value,
;
see https://msdn.microsoft.com/en-us/windows/hardware/gg462963
; Class
= "ActivityMonitor"
;This is determined by the work this filter driver does
; ClassGuid
= {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
;This value is determined by the Load Order Group value
Class = "_TODO_Change_Class_appropriately_"
ClassGuid = {_TODO_Change_ClassGuid_appropriately_}
Provider
= %ManufacturerName%
DriverVer
=
CatalogFile = Sample.cat
```

Директива Signature должна быть установлена на волшебную строку «$ Windows NT $».

Причина этого имени является историческим и не имеет значения для данного обсуждения.

Директивы Class и ClassGuid являются обязательными и определяют класс (тип или группу), к которому драйвер принадлежит.

Сгенерированный INF содержит пример класса ActivityMonitor и связанный с ним GUID, оба в комментариях (комментарии INF указываются после точки с запятой до конца строки).

Самое простое решение - раскомментировать эти строки и удалить фиктивные Class и ClassGuid что приводит к следующему:

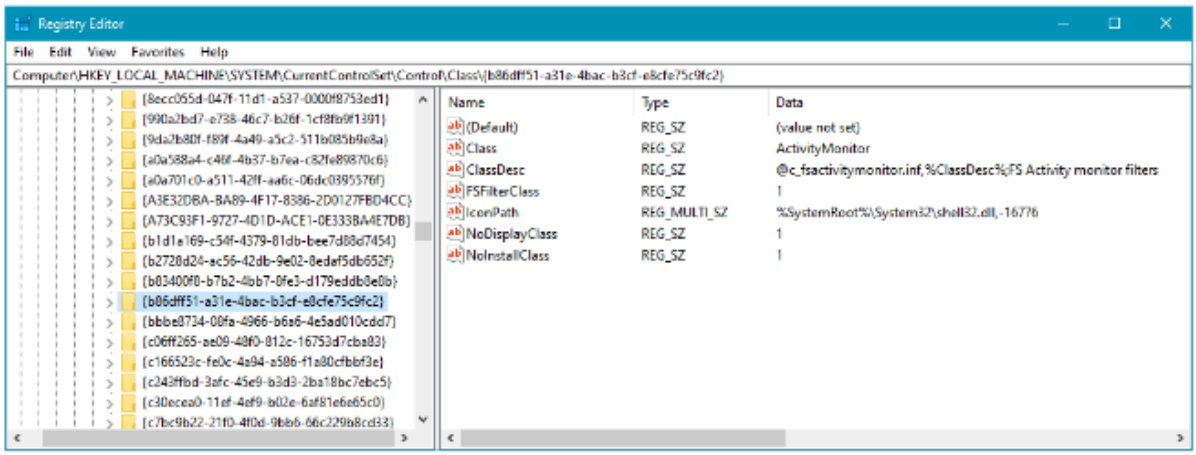
Class = "ActivityMonitor"

ClassGuid = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}

Что это за директива «Класс»? Это одна из предопределенных групп устройств, которые используются в основном аппаратными драйверами, а также мини-фильтрами.

Полный список классов хранится в реестре под ключом HKLM\System\CurrentControlSet\Control\Class. Каждый класс уникально идентифицируется идентификатором GUID.

На рисунке ниже показана запись класса ActivityMonitor в реестре.



Обратите внимание, что GUID - это имя ключа. Само имя класса предоставляется в значении Class. Другие значения в этом ключе не важны с практической точки зрения. Запись, которая делает этот класс «подходящим» для мини-фильтров файловой системы является значение FSFilterClass, имеющее значение 1.

Вы можете просмотреть все существующие классы в системе с помощью инструмента FSClass.exe, доступного в Github репозиторий AllTools (https://github.com/zodiacon/AllTools). Вот пример выполнения:

```
c:\Tools> FSClass.exe
File System Filter Classes version 1.0 (C)2019 Pavel Yosifovich

GUID                                   Name                    Description
--------------------------------------------------------------------------------
{2db15374-706e-4131-a0c7-d7c78eb0289a} SystemRecovery    FS System recovery filters
{3e3f0674-c83c-4558-bb26-9820e1eba5c5} ContentScreener   FS Content screener filters
{48d3ebc4-4cf8-48ff-b869-9c68ad42eb9f} Replication       FS Replication filters
{5d1b9aaa-01e2-46af-849f-272b3f324c46} FSFilterSystem    FS System filters
{6a0a8e78-bba6-4fc4-a709-1e33cd09d67e} PhysicalQuotaManagement  FS Physical quota ma\
nagement filters
{71aa14f8-6fad-4622-ad77-92bb9d7e6947} ContinuousBackup  FS Continuous backup filters
{8503c911-a6c7-4919-8f79-5028f5866b0c} QuotaManagement   FS Quota management filters
{89786ff1-9c12-402f-9c9e-17753c7f4375} CopyProtection    FS Copy protection filters
{a0a701c0-a511-42ff-aa6c-06dc0395576f} Encryption        FS Encryption filters
{b1d1a169-c54f-4379-81db-bee7d88d7454} AntiVirus         FS Anti-virus filters
{b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2} ActivityMonitor   FS Activity monitor filters
{cdcf0939-b75b-4630-bf76-80f7ba655884} CFSMetadataServer FS CFS metadata server filt\
ers
{d02bc3da-0c8e-4945-9bd5-f1883c226c8c} SecurityEnhancer  FS Security enhancer filters
{d546500a-2aeb-45f6-9482-f4b1799c3177} HSM               FS HSM filters
{e55fa6f9-128c-4d04-abab-630c74b1453a} Infrastructure    FS Infrastructure filters
{f3586baf-b5aa-49b5-8d6c-0569284c639f} Compression       FS Compression filters
{f75a86c0-10d8-4c3a-b233-ed60e4cdfaac} Virtualization    FS Virtualization filters
{f8ecafa6-66d1-41a5-899b-66585d7216b7} OpenFileBackup    FS Open file backup filters
{fe8f1572-c67a-48c0-bbac-0b5c6d66cafb} Undelete          FS Undelete filters
```

Вернемся к разделу Version в INF - директива Provider - это имя издателя драйвера.

С практической точки зрения это не имеет большого значения, но может появиться в каком-либо пользовательском интерфейсе, поэтому должно быть что-то значимое. Значение, установленное шаблоном WDK, -% ManufacturerName%.

Все в пределах процентов - это своего рода макрос, который должен быть заменен фактическим значением, указанным в другом разделе, называемом Strings. Вот часть этого раздела:

```
[Strings]
; TODO - Add your manufacturer
ManufacturerName        = "Template"
ServiceDescription      = "Sample Mini-Filter Driver"
ServiceName             = "Sample"
```

Здесь ManufacturerName заменяется на «Template». Автор драйвера должен заменить «Шаблон»с правильным названием компании или продукта, в зависимости от того, что имеет смысл.

Директива DriverVer указывает дату/время драйвера и версию.

CatalogFile - директива указывает на файл каталога, в котором хранятся цифровые подписи и пакет драйвера (обычно файлы SYS, INF и CAT).

Раздел DefaultInstall

Раздел DefaultInstall указывает, какие операции должны выполняться в момент установки драйвера.

Имея этот раздел, драйвер можно установить с помощью проводника Windows, щелкнув правой кнопкой мыши INF и выбрав «Установить».

Мастер WDK создал это так:

[DefaultInstall]

OptionDesc = %ServiceDescription%

CopyFiles = MiniFilter.DriverFiles

Директива OptionDesc предоставляет простое описание, которое используется в случае, если пользователь устанавливает драйвер с помощью мастера установки драйвера Plug&Play (редко используется для мини-фильтров файловой системы).

Важной директивой является CopyFiles, которая указывает какие файлы нужно копировать и куда.

Директива CopyFile указывает на показанный раздел MiniFilter.DriverFiles:

[MiniFilter.DriverFiles]

%DriverName%.sys

% DriverName% указывает на Sample.sys, выходной файл драйвера. Этот файл нужно скопировать, но куда?

В разделе DestinationDirs представлена эта информация:

```
[DestinationDirs]
DefaultDestDir          = 12
MiniFilter.DriverFiles  = 12    ;%windir%\system32\drivers
```

DefaultDestDir - это каталог по умолчанию для копирования, если не указан явно. Значение здесь странное (12), но на самом деле это магическое число, указывающее на каталог системных драйверов, показанное в комментарии для второй директивы. Каталог System32\Drivers - это каноническое место для размещения драйверов. В предыдущих главах мы размещали наши драйверы где угодно, но драйверы должны быть помещены в этлй папке с драйверами, по крайней мере, в целях защиты, так как эта папка является системной и поэтому не разрешает полный доступ для обычных пользователей.

Вот описание магических чисел:

• 10 - Windows directory (%SystemRoot%).

• 11 - System directory (%SystemRoot%\System32).

• 24 - root directory of system disk (e.g. C:\).

• 01 - the directory from which the INF file is read from.

Вторая директива, которую мы видим, на самом деле является названием раздела, который мы только что рассмотрели, MiniFilter.DriverFiles, по существу указание перечисленных там файлов для копирования в целевой каталог.

The Service Section

Следующие интересующие разделы связаны с установкой в разделе реестра служб, аналогично тому, что CreateService делает. Этот шаг является обязательным для любого драйвера. Сначала мы видим такое определение:

[DefaultInstall.Services]

AddService = %ServiceName%,,MiniFilter.Service

К DefaultInstall добавляется «.Services», и такой раздел ищется автоматически.

Если найдена директива AddService, который указывает, какую информацию записывать в реестр в служебном ключе с именем% ServiceName%.

Дополнительная запятая - это заполнитель для флагов, в данном случае ноль (отсюда и запятая).

Код, сгенерированный мастером, выглядит следующим образом:

```
[MiniFilter.Service]
DisplayName      = %ServiceName%
Description      = %ServiceDescription%
ServiceBinary    = %12%\%DriverName%.sys      ;%windir%\system32\drivers\
Dependencies     = "FltMgr"
ServiceType      = 2                          ;SERVICE_FILE_SYSTEM_DRIVER
StartType        = 3                          ;SERVICE_DEMAND_START
ErrorControl     = 1                          ;SERVICE_ERROR_NORMAL
; TODO - Change the Load Order Group value
; LoadOrderGroup = "FSFilter Activity Monitor"
LoadOrderGroup = "_TODO_Change_LoadOrderGroup_appropriately_"
AddReg           = MiniFilter.AddRegistry
```

ServiceType - это 2 для драйвера, связанного с файловой системой (в отличие от 1 для «стандартных» драйверов).

Зависимости - это то, чего мы раньше не встречали - это список служб/драйверов, от которых зависит эта служба/драйвер.

В случае мини-фильтра файловой системы это сам диспетчер фильтров.

LoadOrderGroup следует указывать на основе имен групп мини-фильтров.

В заключение, директива AddReg указывает на другой раздел с инструкциями по добавлению дополнительных записей в реестр.

Вот исправленный раздел MiniFilter.Service:

```
[MiniFilter.Service]
DisplayName      = %ServiceName%
Description      = %ServiceDescription%
ServiceBinary    = %12%\%DriverName%.sys      ;%windir%\system32\drivers\
Dependencies     = "FltMgr"
ServiceType      = 2                          ;SERVICE_FILE_SYSTEM_DRIVER
StartType        = 3                          ;SERVICE_DEMAND_START
ErrorControl     = 1                          ;SERVICE_ERROR_NORMAL
LoadOrderGroup = "FSFilter Activity Monitor"
AddReg           = MiniFilter.AddRegistry
```

Разделы AddReg

Этот раздел (или разделы, их может быть любое количество) используется для добавления пользовательских записей в реестр.

Созданный мастером INF содержит следующие дополнения к реестру:

```
[MiniFilter.AddRegistry]
HKR,,"DebugFlags",0x00010001 ,0x0
HKR,,"SupportedFeatures",0x00010001,0x3
HKR,"Instances","DefaultInstance",0x00000000,%DefaultInstance%
HKR,"Instances\"%Instance1.Name%,"Altitude",0x00000000,%Instance1.Altitude%
HKR,"Instances\"%Instance1.Name%,"Flags",0x00010001,%Instance1.Flags%
```

Синтаксис для каждой записи содержит следующее по порядку:

• корневой ключ - один из HKLM, HKCU (текущий пользователь), HKCR (классы root), HKU (пользователи) или HKR (относительно вызывающего раздела).

• подключ из корневого ключа (использует сам ключ, если не указан)

• имя значения для установки

• flags - определены многие, по умолчанию - ноль, что указывает на запись значения REG_SZ.

Несколько другие флаги:

- 0x100000 - записать REG_MULTI_SZ

- 0x100001 - записать REG_DWORD

- 0x000001 - записать двоичное значение (REG_BINARY)

- 0x000002 - Не перезаписывать существующее значение

- 0x000008 - добавить значение. Существующее значение должно быть REG_MULTI_SZ

• фактическое значение или значения для записи/добавления

Приведенный выше фрагмент кода устанавливает некоторые значения по умолчанию для мини-фильтров файловой системы. Самая важная ценность — это высота, взятая из %Instance1.Altitude% в разделе строк.

Завершенный INF

Последняя часть, которую нам нужно изменить, - это сама высота в разделе Strings. Вот пример для драйвера:

```
[Strings]
; other entries
;Instances specific information.
DefaultInstance         = "Sample Instance"
Instance1.Name          = "Sample Instance"
Instance1.Altitude      = "360100"
Instance1.Flags         = 0x0              ; Allow all attachments
```

Значение высоты было выбрано из диапазона высот группы монитора активности. В настоящем драйвере это число будет возвращено Microsoft на основе запроса высоты, обсужденного ранее.

Наконец, значение флагов указывает, что драйвер может подключаться к любому тому, но на самом деле драйвер будет запрошен в его обратном вызове установки экземпляра.

Установка драйвера

После того, как файл INF будет правильно изменен и код драйвера скомпилирован, он будет готов к установке. Самый простой способ установки - скопировать пакет драйверов (файлы SYS, INF и CAT) в целевую систему.

Затем щелкните правой кнопкой мыши файл INF в проводнике и выберите «Установить». Это запустит INF, выполнив необходимые операции.

На этом этапе мини-фильтр установлен, и его можно загрузить с помощью инструмента командной строки fltmc.

c:\>fltmc load sample

Обработка операций ввода-вывода

Основная функция мини-фильтра файловой системы - обработка операций ввода-вывода путем реализации обратных вызовов для интересующих операций.

Предварительные операции позволяют мини-фильтру отклонять операции полностью, а постоперация позволяет посмотреть результат операции, а в некоторых случаях - внесение изменений в возвращаемую информацию.

Обратные вызовы перед операцией

Все предоперационные обратные вызовы имеют один и тот же прототип, как показано ниже:

```
FLT_PREOP_CALLBACK_STATUS SomePreOperation (
            _Inout_
            PFLT_CALLBACK_DATA Data,
            _In_
            PCFLT_RELATED_OBJECTS FltObjects,
            _Outptr_
            PVOID *CompletionContext);
```

Во-первых, давайте посмотрим на возможные возвращаемые значения FLT_PREOP_CALLBACK_STATUS.

Вот стандартные возвращаемые значения:

• FLT_PREOP_COMPLETE указывает, что драйвер завершает операцию. Менеджер фильтров не вызывает обратный вызов после операции (если зарегистрирован) и не пересылает запрос на мини-фильтры нижнего уровня.

• FLT_PREOP_SUCCESS_NO_CALLBACK указывает, что предварительная операция выполнена с запросом и позволяет ему продолжить переход к следующему фильтру.

• FLT_PREOP_SUCCESS_WITH_CALLBACK указывает, что драйвер позволяет диспетчеру фильтров распространять блокировать запрос фильтрам нижнего уровня, но он хочет, чтобы его пост-обратный вызов был вызван для этой операции.

• FLT_PREOP_PENDING указывает, что драйвер ожидает выполнения операции. Менеджер фильтров не продолжает обработку запроса, пока драйвер не вызовет FltCompletePendedPreOperation.

• FLT_PREOP_SYNCHRONIZE похож на FLT_PREOP_SUCCESS_WITH_CALLBACK, но драйвер просит диспетчер фильтров вызвать его пост-обратный вызов в том же потоке на IRQL <= APC_LEVEL (обычно обратный вызов после операции может быть вызван на IRQL <= DISPATCH_LEVEL).

Аргумент Data предоставляет всю информацию, относящуюся к самой операции ввода-вывода, в виде FLT_CALLBACK_DATA структуры:

```
typedef struct _FLT_CALLBACK_DATA {
	FLT_CALLBACK_DATA_FLAGS Flags;
	PETHREAD CONST Thread;
	PFLT_IO_PARAMETER_BLOCK CONST Iopb;
	IO_STATUS_BLOCK IoStatus;
	struct _FLT_TAG_DATA_BUFFER *TagData;
	union {
		struct {
			LIST_ENTRY QueueLinks;
			PVOID QueueContext[2];
		};
		PVOID FilterContext[4];
	};
	KPROCESSOR_MODE RequestorMode;
} FLT_CALLBACK_DATA, *PFLT_CALLBACK_DATA;
```

Эта структура также предоставляется в пост-обратном вызове. Вот краткое изложение важных членов этой структуры:

• Флаги могут содержать ноль или комбинацию флагов, некоторые из которых перечислены ниже:

- FLTFL_CALLBACK_DATA_DIRTY указывает, что драйвер внес изменения в структуру и затем вызвал FltSetCallbackDataDirty. Каждый член структуры может быть изменен кроме Thread и RequestorMode.

- FLTFL_CALLBACK_DATA_FAST_IO_OPERATION указывает, что это быстрая операция ввода-вывода.

- FLTFL_CALLBACK_DATA_IRP_OPERATION указывает, что это операция на основе IRP.

- FLTFL_CALLBACK_DATA_GENERATED_IO указывает, что это операция, сгенерированная другим мини-фильтром.

- FLTFL_CALLBACK_DATA_POST_OPERATION указывает, что это постоперация.

• Thread - Указатель на поток, запрашивающий эту операцию.

• IoStatus - Статус запроса. Предварительная операция может установить это значение, а затем указать, что операция завершается возвратом FLT_PREOP_COMPLETE. После операции можно посмотреть на окончательный статус операции.

• RequestorMode указывает, находится ли запросчик операции из пользовательского режима (UserMode) или режим ядра (KernelMode).

• Iopb сам по себе является структурой, содержащей подробные параметры запроса, определенные следующим образом:

ULONG IrpFlags;

```
UCHAR MajorFunction;

UCHAR MinorFunction;

UCHAR OperationFlags;

UCHAR Reserved;

PFILE_OBJECT TargetFileObject;

PFLT_INSTANCE TargetInstance;

FLT_PARAMETERS Parameters;

} FLT_IO_PARAMETER_BLOCK, *PFLT_IO_PARAMETER_BLOCK;
```

Полезными членами этой структуры являются следующие:

• TargetFileObject - файловый объект, являющийся целью данной операции.

• Parameters - это объединение отделяющий фактические данные для конкретной информации (аналогичные в концепции члену Paramters IO_STACK_LOCATION). Драйвер смотрит на правильную структуру внутри этого объединения, чтобы получить необходимую информацию.

Мы рассмотрим некоторые из этих структур, когда мы рассмотрим конкретные типы операций далее в этой главе.

Второй аргумент предварительного обратного вызова - это другая структура типа FLT_RELATED_OBJECTS.

Эта структура в основном содержит дескрипторы текущего фильтра, экземпляра и тома, которые полезны в некоторых API.

Вот полное определение этой структуры:

```
typedef struct _FLT_RELATED_OBJECTS {

      USHORT CONST Size;

      USHORT CONST TransactionContext;

      PFLT_FILTER CONST Filter;

      PFLT_VOLUME CONST Volume;

      PFLT_INSTANCE CONST Instance;

      PFILE_OBJECT CONST FileObject;
```

```
    PKTRANSACTION CONST Transaction;
} FLT_RELATED_OBJECTS, *PFLT_RELATED_OBJECTS;
```

Поле FileObject совпадает с полем, доступным через TargetFileObject блока параметров ввода-вывода..

Последний аргумент предварительного обратного вызова - это значение контекста, которое может быть установлено драйвером.

Если установлено, это значение распространяется на подпрограмму пост-обратного вызова для того же запроса (значение по умолчанию - NULL).

Обратные вызовы после операции

Все обратные вызовы после операции имеют один и тот же прототип, как показано ниже:

```
FLT_POSTOP_CALLBACK_STATUS SomePostOperation (
            _Inout_ PFLT_CALLBACK_DATA Data,
            _In_ PCFLT_RELATED_OBJECTS FltObjects,
            _In_opt_ PVOID CompletionContext,
            _In_ FLT_POST_OPERATION_FLAGS Flags);
```

Послеоперационная функция вызывается в IRQL <= DISPATCH_LEVEL в произвольном контексте потока, если процедура предварительного обратного вызова не вернула FLT_PREOP_SYNCHRONIZE, и в этом случае диспетчер фильтров гарантирует, что пост-обратный вызов вызывается при IRQL <DISPATCH_LEVEL в том же потоке, который выполнялся предварительным обратным вызовом.

В этом случае драйвер не может выполнять определенные типы операций, потому что IRQL слишком высоко:

• Нет доступа к выгружаемой памяти.

• Невозможно использовать API ядра, которые работают только при IRQL < DISPATCH_LEVEL.

• Невозможно получить примитивы синхронизации, такие как мьютексы, быстрые мьютексы, исполнительные ресурсы, семафоры, события и т. д. (Однако он может получить спин-блокировки).

• Невозможно установить, получить или удалить контексты (см. Раздел «Контексты» далее в этой главе), но он может контексты выпуска.

Если драйверу нужно сделать что-либо из вышеперечисленного, он каким-то образом должен отложить его выполнение до другой процедуры.

Это можно сделать одним из двух способов:

• Драйвер вызывает FltDoCompletionProcessingWhenSafe, который устанавливает функцию обратного вызова, которая вызывается системным рабочим потоком на IRQL < DISPATCH_LEVEL.

• Драйвер публикует рабочий элемент, вызывая FltQueueDeferredIoWorkItem, который ставит работу в очередь, элемент которой в конечном итоге будет выполняться системным рабочим потоком при IRQL = PASSIVE_LEVEL.

Хотя использовать FltDoCompletionProcessingWhenSafe проще, у него есть некоторые ограничения:

• Не может использоваться для IRP_MJ_READ, IRP_MJ_WRITE или IRP_MJ_FLUSH_BUFFERS.

• Может вызываться только для операций на основе IRP (можно проверить с помощью макроса FLT_IS_IRP_OPERATION).

Возвращаемое значение из обратного вызова обычно равно FLT_POSTOP_FINISHED_PROCESSING.

Убедитесь, что драйвер завершил эту операцию. Однако драйвер может вернуть FLT_POSTOP_MORE_PROCESSING_REQUIRED, чтобы сообщить файловому менеджеру, что операция все еще ожидает завершения.

А затем вызвать FltCompletePendedPostOperation, чтобы диспетчер фильтров знал, что он может продолжить обработку этого запроса.

Драйвер защиты от удаления

Пришло время поместить часть информации, о которой уже говорилось, в настоящий драйвер.

Мы создадим драйвер, который сможет защитить определенные файлы от удаления определенными процессами.

Мы построим драйвер на основе шаблона проекта, предоставленного WDK (хотя мне не нравится часть сгенерированного кода по этому шаблону).

Мы начнем с создания нового проекта мини-фильтра файловой системы с именем DelProtect (или другим именем по вашему выбору) и позвольте мастеру сгенерировать исходные файлы и код.

Далее займемся файлом INF. Этот драйвер будет принадлежать классу «Отменить удаление» (кажется, разумно), и мы выберем высоту в этом диапазоне. Вот измененные разделы в INF:

```
[Version]
Signature   = "$Windows NT$"
Class       = "Undelete"
ClassGuid   = {fe8f1572-c67a-48c0-bbac-0b5c6d66cafb}
Provider    = %ManufacturerName%
DriverVer   =
CatalogFile = DelProtect.cat

[MiniFilter.Service]
DisplayName      = %ServiceName%
Description      = %ServiceDescription%
ServiceBinary    = %12%\%DriverName%.sys        ;%windir%\system32\drivers\
Dependencies     = "FltMgr"
ServiceType      = 2                            ;SERVICE_FILE_SYSTEM_DRIVER
StartType        = 3                            ;SERVICE_DEMAND_START
ErrorControl     = 1                            ;SERVICE_ERROR_NORMAL
LoadOrderGroup = "FS Undelete filters"
AddReg           = MiniFilter.AddRegistry

[Strings]
ManufacturerName        = "WindowsDriversBook"
ServiceDescription      = "DelProtect Mini-Filter Driver"
ServiceName             = "DelProtect"
DriverName              = "DelProtect"
DiskId1                 = "DelProtect Device Installation Disk"

;Instances specific information.
DefaultInstance         = "DelProtect Instance"
Instance1.Name          = "DelProtect Instance"
Instance1.Altitude       = "345101"    ; in the range of the undelete group
Instance1.Flags         = 0x0         ; Allow all attachments
```

Теперь, когда мы закончили с INF, мы можем обратить внимание на код.

Созданный исходный файл называется DelProtect.c, поэтому сначала мы переименуем его в DelProtect.cpp, чтобы мы могли свободно использовать C ++.

DriverEntry, предоставленный шаблоном проекта, уже имеет регистрационный код мини-фильтра. Нам нужно настроить обратные вызовы, чтобы указать, какие именно из них нам действительно интересны.

Итак вопрос в том, какие основные функции задействованы при удалении файлов ?

Оказывается, есть два способа удалить файл. Один из способов - использовать IRP_MJ_SET_INFORMATION. Эта операция предоставляет набор операций, удаление является лишь одной из них.

Второй способ для удаления файла (и на самом деле наиболее часто используемый), нужно открыть файл с помощью FILE_DELETE_ON_CLOSE. Затем файл удаляется, как только закрывается его последний дескриптор.

Что касается драйвера, мы хотим поддержать оба варианта удаления.

Изменим массив FLT_OPERATION_REGISTRATION:

```
CONST FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE, 0, DelProtectPreCreate, nullptr },
    { IRP_MJ_SET_INFORMATION, 0, DelProtectPreSetInformation, nullptr },
    { IRP_MJ_OPERATION_END }
};
```

Конечно, нам необходимо реализовать соответствующие DelProtectPreCreate и DelProtectPreSetInformation.

Оба являются предварительными обратными вызовами, поскольку мы хотим отклонить эти запросы при определенных условиях.

Обработка предварительного создания

Начнем с функции предварительного создания (которая несколько проще). Его прототип такой же, как любой предварительный обратный вызов (просто скопируйте его из общего DelProtectPreOperation, предоставленного проектом):

```
FLT_PREOP_CALLBACK_STATUS DelProtectPreCreate(
            _Inout_ PFLT_CALLBACK_DATA Data,
            _In_ PCFLT_RELATED_OBJECTS FltObjects,
            _Flt_CompletionContext_Outptr_ PVOID *CompletionContext);
```

Сначала мы проверим, исходит ли операция из режима ядра, и если да, просто позвольте ей продолжить, что-бы не было проблем:

```
UNREFERENCED_PARAMETER(CompletionContext);
UNREFERENCED_PARAMETER(FltObjects);
if (Data->RequestorMode == KernelMode)
      return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Затем нам нужно проверить, существует ли флаг FILE_DELETE_ON_CLOSE в запросе на создание. В структуре, на которую нужно смотреть - это поле Create под Paramaters внутри Iopb, например:

```
const auto& params = Data->Iopb->Parameters.Create;
if (params.Options & FILE_DELETE_ON_CLOSE) {
	// delete operation
}
// otherwise, just carry on
return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Вышеупомянутая переменная params ссылается на структуру Create, определенную следующим образом:

```
struct {
	PIO_SECURITY_CONTEXT SecurityContext;
	//
	// The low 24 bits contains CreateOptions flag values.
	// The high 8 bits contains the CreateDisposition values.
	//
	ULONG Options;
	USHORT POINTER_ALIGNMENT FileAttributes;
	USHORT ShareAccess;
	ULONG POINTER_ALIGNMENT EaLength;
	PVOID EaBuffer;
	LARGE_INTEGER AllocationSize;
} Create;
```

Как правило, для любой операции ввода-вывода необходимо обращаться к документации, чтобы понять, что доступно и как пользоваться.

В нашем случае поле `Options` представляет собой комбинацию задокументированных флагов.

Под функцией FltCreateFile (которую мы будем использовать позже в этой главе), код проверяет, существует ли этот флаг, и если это так, это означает, что инициируется операция удаления.

Для этого драйвера мы заблокируем операции удаления, исходящие от процессов cmd.exe. Для этого нам нужно чтобы получить путь к отображению вызывающего процесса. Поскольку операция создания вызывается синхронно, мы знаем, что вызывающий процесс - это процесс, пытающийся что-то удалить. Но как нам получить отображение и путь текущего процесса ?

Один из способов - использовать NtQueryInformationProcess из ядра (или его эквивалент Zw - ZwQueryInformationProcess). Он частично задокументирован, и его прототип доступен в заголовке модели пользователя <wintrnl.h>.

Мы можем просто скопируем его объявление:

```
extern "C" NTSTATUS ZwQueryInformationProcess(

            _In_

            HANDLE

            ProcessHandle,

            _In_

            PROCESSINFOCLASS ProcessInformationClass,

            _Out_

            PVOID

            ProcessInformation,

            _In_

            ULONG

            ProcessInformationLength,

            _Out_opt_ PULONG

            ReturnLength);
```

Перечисление PROCESSINFOCLASS в основном доступно в <ntddk.h>. Я говорю «в основном», потому что не предоставляет все поддерживаемые значения. (Мы вернемся к этому вопросу в главе 11).

Для целей нашего драйвера мы можем использовать значение ProcessImageFileName из PROCESSINFOCLASS. С его помощью мы можем получить полный путь к файлу образа процесса.

В документации для NtQueryInformationProcess указано, что для ProcessImageFileName, возвращенные данные представляют собой структуру UNICODE_STRING, которая должна быть выделена вызывающей стороной:

```
auto size = 300;// some arbitrary size enough for cmd.exe image path

auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);

if (processName == nullptr)

      return FLT_PREOP_SUCCESS_NO_CALLBACK;

RtlZeroMemory(processName, size);// ensure string will be NULL-terminated
```

Теперь мы можем сделать вызов API:

```
auto status = ZwQueryInformationProcess(NtCurrentProcess(),
ProcessImageFileName,
            processName, size - sizeof(WCHAR), nullptr);
```

Макрос NtCurrentProcess возвращает дескриптор, который относится к текущему процессу (на практике практически то же самое, что и API пользовательского режима GetCurrentProcess).

Если вызов завершился успешно, нам нужно сравнить имя файла образа процесса с cmd.exe. Вот один простой способ сделать это:

```
if (NT_SUCCESS(status)) {
      if (wcsstr(processName->Buffer, L"\\System32\\cmd.exe") != nullptr ||
                  wcsstr(processName->Buffer, L"\\SysWOW64\\cmd.exe") !=
nullptr) {
            // do something
      }
}
```

Сравнение не такое простое, как хотелось бы. Фактический путь, возвращенный ZwQueryInformationProcess - это собственный путь, например «\Device\HarddiskVolume3\Windows\System32\cmd.exe».

Для этого простого драйвера мы просто ищем подстроку «System32\cmd.exe» или «SysWOW64\cmd.exe» (последнее - в случае вызова 32-битного cmd.exe). Кстати, сравнение чувствительно к регистру.

Это не идеально. Что, если кто-то скопирует cmd.exe в другую папку и запустит его оттуда ? Это действительно зависит от драйвера. Мы могли бы так же легко сравнить с cmd.exe только. Для этого базового драйвера этого достаточно.

Если это действительно cmd.exe, нам нужно предотвратить успешное выполнение операции. Стандартный способ, это изменить статус операции (Data-> IoStatus.Status) на соответствующую ошибку и вернуться из обратного вызова FLT_PREOP_COMPLETE, чтобы сообщить диспетчеру фильтров не продолжать операцию.

Вот немного измененный обратный вызов перед созданием:

```
_Use_decl_annotations_

FLT_PREOP_CALLBACK_STATUS DelProtectPreCreate(

            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
PVOID*) {

      UNREFERENCED_PARAMETER(FltObjects);

      if (Data->RequestorMode == KernelMode)

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      auto& params = Data->Iopb->Parameters.Create;

      auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;

      if (params.Options & FILE_DELETE_ON_CLOSE) {

            // delete operation

            KdPrint(("Delete on close: %wZ\n", &Data->Iopb->TargetFileObject-
>FileName));

            auto size = 300;

            // some arbitrary size enough for cmd.exe image path

            auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool,
size);

            if (processName == nullptr)

                  return FLT_PREOP_SUCCESS_NO_CALLBACK;

            RtlZeroMemory(processName, size);

            // ensure string will be NULL-terminated

            auto status = ZwQueryInformationProcess(NtCurrentProcess(),

                        ProcessImageFileName, processName, size -
sizeof(WCHAR), nullptr);

            if (NT_SUCCESS(status)) {

                  KdPrint(("Delete operation from %wZ\n", processName));

                  if (wcsstr(processName->Buffer, L"\\System32\\cmd.exe") !=
nullptr ||

                              wcsstr(processName->Buffer,
L"\\SysWOW64\\cmd.exe") != nullptr) {

                        // fail request

                        Data->IoStatus.Status = STATUS_ACCESS_DENIED;

                        returnStatus = FLT_PREOP_COMPLETE;
```

```
                        KdPrint(("Prevent delete from IRP_MJ_CREATE by
cmd.exe\n"));

                }

            }

            ExFreePool(processName);

      }

      return returnStatus;

}
```

Чтобы собрать драйвер, нам нужно предоставить что-то для предварительного обратного вызова IRP_MJ_SET_INFORMATION.

Вот простая реализация, разрешающая все:

```
FLT_PREOP_CALLBACK_STATUS DelProtectPreSetInformation(

            _Inout_ PFLT_CALLBACK_DATA Data, _In_ PCFLT_RELATED_OBJECTS
FltObjects, PVOID*) {

      UNREFERENCED_PARAMETER(FltObjects);

      UNREFERENCED_PARAMETER(Data);

      return FLT_PREOP_SUCCESS_NO_CALLBACK;

}
```

Если мы соберем и запустим драйвер, а затем протестируем его примерно так:

del somefile.txt

Мы увидим, что, хотя мы добираемся до нашего обработчика IRP_MJ_CREATE и не выполняем запрос, файл все еще успешно удален. Причина этого в том, что cmd.exe несколько подлый, и если он не работает в одном направлении, он пытается другой.

После сбоя использования FILE_DLETE_ON_CLOSE используется IRP_MJ_SET_INFORMATION, и поскольку мы разрешаем выполнение всех операций для IRP_MJ_SET_INFORMATION - операция выполняется успешно.)

Обработка предустановленной информации

Второй способ удаления файлов осуществляется файловыми системами. Мы начнем с игнорирования вызывающих с IRP_MJ_CREATE:

```
_Use_decl_annotations_

FLT_PREOP_CALLBACK_STATUS DelProtectPreSetInformation(

            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
PVOID*) {

      UNREFERENCED_PARAMETER(FltObjects);

      if (Data->RequestorMode == KernelMode)

            return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Поскольку IRP_MJ_SET_INFORMATION - это способ выполнять несколько типов операций, нам нужно проверить, что это фактически операция удаления.

Драйвер должен сначала получить доступ к правильной структуре в параметрах объединения, объявленной так:

```
struct {

      ULONG Length;

      FILE_INFORMATION_CLASS POINTER_ALIGNMENT FileInformationClass;

      PFILE_OBJECT ParentOfTarget;

      union {

            struct {

                  BOOLEAN ReplaceIfExists;

                  BOOLEAN AdvanceOnly;

            };

            ULONG ClusterCount;

            HANDLE DeleteHandle;

      };

      PVOID InfoBuffer;

} SetFileInformation;
```

FileInformationClass указывает, какой тип операции представляет этот экземпляр, поэтому нам нужно проверить, является ли это операцией удаления:

```
auto& params = Data->Iopb->Parameters.SetFileInformation;
if (params.FileInformationClass != FileDispositionInformation &&
            params.FileInformationClass != FileDispositionInformationEx) {
      // not a delete operation
      return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

Значение перечисления FileDispositionInformation указывает на операцию удаления.

FileDispositionInformationEx аналогичен и недокументирован, но используется внутри пользовательского режима DeleteFile, поэтому мы проверяем и то, и другое.

Если это операция удаления, нужно сделать еще одну проверку, посмотрите на информационный буфер который имеет тип FILE_DISPOSITION_INFORMATION для операций удаления:

```
auto info = (FILE_DISPOSITION_INFORMATION*)params.InfoBuffer;
if (!info->DeleteFile)
      return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Наконец, мы находимся в операции удаления. В случае IRP_MJ_CREATE обратный вызов вызывается запрашивающим поток (и, следовательно, запрашивающий процесс), поэтому мы можем просто получить доступ к текущему процессу для того чтобы узнать, какой файл отображения используется для вызова.

```
// what process did this originate from?
auto process = PsGetThreadProcess(Data->Thread);
NT_ASSERT(process);
// cannot really fail
```

Наша цель - вызвать ZwQueryInformationProcess, но для этого нам нужен дескриптор. Функция ObOpenObjectByPointer позволяет получить дескриптор объекта.

Это определено вот так:

```
NTSTATUS ObOpenObjectByPointer(
            _In_ PVOID Object,
            _In_ ULONG HandleAttributes,
            _In_opt_ PACCESS_STATE PassedAccessState,
            _In_ ACCESS_MASK DesiredAccess,
            _In_opt_ POBJECT_TYPE ObjectType,
            _In_ KPROCESSOR_MODE AccessMode,
```

```
            _Out_ PHANDLE Handle);
```

Аргументы ObOpenObjectByPointer описаны ниже:

• `Object` - это объект, для которого требуется хендл. Это может быть любой тип объекта ядра.

• HandleAttributes - это набор необязательных флагов. Самый полезный флаг - OBJ_KERNEL_HANDLE (мы обсудим другие флаги в главе 11). Этот флаг делает возвращаемый дескриптор дескриптором ядра, который не может использоваться кодом пользовательского режима и может использоваться из любого контекста процесса.

• PassedAccessState - это необязательный указатель на структуру ACCESS_STATE, обычно не полезный для драйверов - установить в NULL.

• DesiredAccess - это маска доступа, с которой должен открываться дескриптор.

• ObjectType - это необязательный тип объекта, с которым функция может сравнивать объект.

• AccessMode может быть UserMode или KernelMode. Драйверы обычно указывают KernelMode. В режиме KernelMode проверка доступа не выполняется.

• Handle - указатель на возвращаемый дескриптор.

Учитывая указанную выше функцию, открытие дескриптора процесса происходит следующим образом:

```
HANDLE hProcess;
auto status = ObOpenObjectByPointer(process, OBJ_KERNEL_HANDLE, nullptr, 0,
            nullptr, KernelMode, &hProcess);
if (!NT_SUCCESS(status))
      return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Получив дескриптор процесса, мы можем запросить имя файла образа процесса и посмотреть, является ли это cmd.exe:

```
auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
auto size = 300;
auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
if (processName) {
      RtlZeroMemory(processName, size);
      // ensure string will be NULL-terminated
      status = ZwQueryInformationProcess(hProcess, ProcessImageFileName,
                  processName, size - sizeof(WCHAR), nullptr);
```

```
	if (NT_SUCCESS(status)) {
		KdPrint(("Delete operation from %wZ\n", processName));
		if (wcsstr(processName->Buffer, L"\\System32\\cmd.exe") != 
nullptr ||
			wcsstr(processName->Buffer, L"\\SysWOW64\\cmd.exe") ! 
= nullptr) {
			Data->IoStatus.Status = STATUS_ACCESS_DENIED;
			returnStatus = FLT_PREOP_COMPLETE;
			KdPrint(("Prevent delete from IRP_MJ_SET_INFORMATION by 
cmd.exe\n"));
		}
	}
	ExFreePool(processName);
}
ZwClose(hProcess);
return returnStatus;
}
```

Теперь мы можем протестировать полный драйвер - мы обнаружим, что cmd.exe не может удалять файлы - «доступ запрещен»- это возвращенная ошибка.

Некоторый рефакторинг

Два реализованных нами предварительных обратных вызывов имеют много общего кода, поэтому следуя принципу DRY («Не повторяйся»), мы можем вынести код, который будет открывать дескриптор процесса, получать отображение файла и сравнивать с cmd.exe отдельной функцией:

```
bool IsDeleteAllowed(const PEPROCESS Process) {
	bool currentProcess = PsGetCurrentProcess() == Process;
	HANDLE hProcess;
	if (currentProcess)
		hProcess = NtCurrentProcess();
	else {
		auto status = ObOpenObjectByPointer(Process, OBJ_KERNEL_HANDLE,
				nullptr, 0, nullptr, KernelMode, &hProcess);
		if (!NT_SUCCESS(status))
			return true;
	}
	auto size = 300;
	bool allowDelete = true;
	auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
	if (processName) {
		RtlZeroMemory(processName, size);
		auto status = ZwQueryInformationProcess(hProcess,
ProcessImageFileName,
				processName, size - sizeof(WCHAR), nullptr);
		if (NT_SUCCESS(status)) {
			KdPrint(("Delete operation from %wZ\n", processName));
			if (wcsstr(processName->Buffer, L"\\System32\\cmd.exe") !=
nullptr ||
					wcsstr(processName->Buffer,
L"\\SysWOW64\\cmd.exe") != nullptr) {
				allowDelete = false;
			}
		}
		ExFreePool(processName);
	}
	if (!currentProcess)
		ZwClose(hProcess);
```

```
	return allowDelete;
}
```

Функция принимает указатель на процесс, пытающийся удалить файл.

Теперь мы можем подключить вызов этой функции к обработке IRP_MJ_CREATE. Вот исправленная функция:

```
_Use_decl_annotations_
FLT_PREOP_CALLBACK_STATUS DelProtectPreCreate(
		PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
PVOID*) {
	UNREFERENCED_PARAMETER(FltObjects);
	if (Data->RequestorMode == KernelMode)
		return FLT_PREOP_SUCCESS_NO_CALLBACK;
	auto& params = Data->Iopb->Parameters.Create;
	auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
	if (params.Options & FILE_DELETE_ON_CLOSE) {
		// delete operation
		KdPrint(("Delete on close: %wZ\n", &Data->Iopb->TargetFileObject-
>FileName));
		if (!IsDeleteAllowed(PsGetCurrentProcess())) {
			Data->IoStatus.Status = STATUS_ACCESS_DENIED;
			returnStatus = FLT_PREOP_COMPLETE;
			KdPrint(("Prevent delete from IRP_MJ_CREATE by
cmd.exe\n"));
		}
	}
	return returnStatus;
}
```

И исправленный предварительный обратный вызов IRP_MJ_SET_INFORMATION:

```
FLT_PREOP_CALLBACK_STATUS DelProtectPreSetInformation(
            _Inout_ PFLT_CALLBACK_DATA Data, _In_ PCFLT_RELATED_OBJECTS
FltObjects, PVOID*) {
      UNREFERENCED_PARAMETER(FltObjects);
      UNREFERENCED_PARAMETER(Data);
      auto& params = Data->Iopb->Parameters.SetFileInformation;
      if (params.FileInformationClass != FileDispositionInformation &&
                  params.FileInformationClass !=
FileDispositionInformationEx) {
            // not a delete operation
            return FLT_PREOP_SUCCESS_NO_CALLBACK;
      }
      auto info = (FILE_DISPOSITION_INFORMATION*)params.InfoBuffer;
      if (!info->DeleteFile)
            return FLT_PREOP_SUCCESS_NO_CALLBACK;
      auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
      // what process did this originate from?
      auto process = PsGetThreadProcess(Data->Thread);
      NT_ASSERT(process);
      if (!IsDeleteAllowed(process)) {
            Data->IoStatus.Status = STATUS_ACCESS_DENIED;
            returnStatus = FLT_PREOP_COMPLETE;
            KdPrint(("Prevent delete from IRP_MJ_SET_INFORMATION by
cmd.exe\n"));
      }
      return returnStatus;
}
```

Обобщение драйвера

Текущий драйвер проверяет только операции удаления из cmd.exe. Обобщим драйвер так, чтобы можно-было зарегистрировать имена исполняемых файлов, для которых можно предотвратить операции удаления.

Для этого мы создадим «классический» объект устройства и символическую ссылку, как это было сделано ранее.

Это не проблема, и драйвер может выполнять двойную функцию: мини-фильтр файловой системы и выставить объект управляющего устройства (CDO).

Мы будем управлять именами исполняемых файлов в массиве фиксированного размера для простоты.

А еще будем использовать созданные нами обертки - FastMutex и AutoLock.

Вот добавленные глобальные переменные:

```
const int MaxExecutables = 32;

WCHAR* ExeNames[MaxExecutables];

int ExeNamesCount;

FastMutex ExeNamesLock;
```

Далее нужно связать и настроить процедуры отправки, а также зарегистрироваться в качестве мини-фильтра:

```
PDEVICE_OBJECT DeviceObject = nullptr;

UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\device\\delprotect");

UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\delprotect");

auto symLinkCreated = false;

do {

      status = IoCreateDevice(DriverObject, 0, &devName,

                  FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

      if (!NT_SUCCESS(status))

            break;

      status = IoCreateSymbolicLink(&symLink, &devName);

      if (!NT_SUCCESS(status))

            break;

      symLinkCreated = true;

      status = FltRegisterFilter(DriverObject, &FilterRegistration,
&gFilterHandle);

      FLT_ASSERT(NT_SUCCESS(status));

      if (!NT_SUCCESS(status))

            break;

      DriverObject->DriverUnload = DelProtectUnloadDriver;

      DriverObject->MajorFunction[IRP_MJ_CREATE] =
```

```
                  DriverObject->MajorFunction[IRP_MJ_CLOSE] =
DelProtectCreateClose;

      DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DelProtectDeviceControl;

      ExeNamesLock.Init();

      status = FltStartFiltering(gFilterHandle);

} while (false);

if (!NT_SUCCESS(status)) {

      if (gFilterHandle)

            FltUnregisterFilter(gFilterHandle);

      if (symLinkCreated)

            IoDeleteSymbolicLink(&symLink);

      if (DeviceObject)

            IoDeleteDevice(DeviceObject);

}

return status;
```

Мы определим несколько управляющих кодов ввода-вывода для добавления, удаления и очистки списка имен исполняемых файлов (в новом файле DelProtectCommon.h):

```
#define IOCTL_DELPROTECT_ADD_EXE

\

CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_DELPROTECT_REMOVE_EXE \

CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_DELPROTECT_CLEAR

\

CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

В обработке таких управляющих кодов нет ничего нового

Процедура отправки DEVICE_CONTROL:

```
NTSTATUS DelProtectDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
	auto stack = IoGetCurrentIrpStackLocation(Irp);
	auto status = STATUS_SUCCESS;
	switch (stack->Parameters.DeviceIoControl.IoControlCode) {
	case IOCTL_DELPROTECT_ADD_EXE:
	{
		auto name = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
		if (!name) {
			status = STATUS_INVALID_PARAMETER;
			break;
		}
		if (FindExecutable(name)) {
			break;
		}
		AutoLock locker(ExeNamesLock);
		if (ExeNamesCount == MaxExecutables) {
			status = STATUS_TOO_MANY_NAMES;
			break;
		}
		for (int i = 0; i < MaxExecutables; i++) {
			if (ExeNames[i] == nullptr) {
				auto len = (::wcslen(name) + 1) * sizeof(WCHAR);
				auto buffer =
(WCHAR*)ExAllocatePoolWithTag(PagedPool, len,
					DRIVER_TAG);
				if (!buffer) {
					status = STATUS_INSUFFICIENT_RESOURCES;
					break;
				}
				::wcscpy_s(buffer, len / sizeof(WCHAR), name);
				ExeNames[i] = buffer;
				++ExeNamesCount;
```

```
                    break;
                }
            }
            break;
        }
        case IOCTL_DELPROTECT_REMOVE_EXE:
        {
            auto name = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
            if (!name) {
                status = STATUS_INVALID_PARAMETER;
                break;
            }
            AutoLock locker(ExeNamesLock);
            auto found = false;
            for (int i = 0; i < MaxExecutables; i++) {
                if (::_wcsicmp(ExeNames[i], name) == 0) {
                    ExFreePool(ExeNames[i]);
                    ExeNames[i] = nullptr;
                    --ExeNamesCount;
                    found = true;
                    break;
                }
            }
            if (!found)
                status = STATUS_NOT_FOUND;
            break;
        }
        case IOCTL_DELPROTECT_CLEAR:
            ClearAll();
            break;
        default:
```

```
            status = STATUS_INVALID_DEVICE_REQUEST;
            break;
      }
      Irp->IoStatus.Status = status;
      Irp->IoStatus.Information = 0;
      IoCompleteRequest(Irp, IO_NO_INCREMENT);
      return status;
}
```

В приведенном выше коде отсутствуют вспомогательные функции FindExecutable и ClearAll определяются так:

```
bool FindExecutable(PCWSTR name) {
      AutoLock locker(ExeNamesLock);
      if (ExeNamesCount == 0)
            return false;
      for (int i = 0; i < MaxExecutables; i++)
            if (ExeNames[i] && ::_wcsicmp(ExeNames[i], name) == 0)
                  return true;
      return false;
}
void ClearAll() {
      AutoLock locker(ExeNamesLock);
      for (int i = 0; i < MaxExecutables; i++) {
            if (ExeNames[i]) {
                  ExFreePool(ExeNames[i]);
                  ExeNames[i] = nullptr;
            }
      }
      ExeNamesCount = 0;
}
```

Имея приведенный выше код, нам нужно внести изменения в обратный вызов pre-create для поиска имен исполняемых файлов в массиве, которым мы управляем. Вот исправленный код:

```
_Use_decl_annotations_

FLT_PREOP_CALLBACK_STATUS DelProtectPreCreate(

            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
PVOID*) {

      if (Data->RequestorMode == KernelMode)

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      auto& params = Data->Iopb->Parameters.Create;

      auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;

      if (params.Options & FILE_DELETE_ON_CLOSE) {

            // delete operation

            KdPrint(("Delete on close: %wZ\n", &FltObjects->FileObject-
>FileName));

            auto size = 512;

            // some arbitrary size

            auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool,
size);

            if (processName == nullptr)

                  return FLT_PREOP_SUCCESS_NO_CALLBACK;

            RtlZeroMemory(processName, size);

            auto status = ZwQueryInformationProcess(NtCurrentProcess(),
ProcessImageFile\

                        Name,

                        processName, size - sizeof(WCHAR), nullptr);

            if (NT_SUCCESS(status)) {

                  KdPrint(("Delete operation from %wZ\n", processName));

                  auto exeName = ::wcsrchr(processName->Buffer, L'\\');

                  NT_ASSERT(exeName);

                  if (exeName && FindExecutable(exeName + 1)) {

                        // skip backslash

                        Data->IoStatus.Status = STATUS_ACCESS_DENIED;
```

```
                        KdPrint(("Prevented delete in IRP_MJ_CREATE\n"));
                        returnStatus = FLT_PREOP_COMPLETE;
                    }
                }
                ExFreePool(processName);
            }
            return returnStatus;
}
```

Основная модификация в приведенном выше коде - это вызов FindExecutable, чтобы узнать, находится ли текущий процесс.

Имя исполняемого файла - одно из значений, хранящихся в массиве. Если это так, мы устанавливаем «доступ запрещен».

Тестирование модифицированного драйвера

Ранее мы тестировали драйвер, удаляя файлы с помощью cmd.exe, но это может быть недостаточно универсальным, поэтому лучше создать собственное тестовое приложение.

Есть три способа удалить файл с помощью API пользовательского режима:

1. Вызовите функцию DeleteFile.

2. Вызовите CreateFile с флагом FILE_FLAG_DELETE_ON_CLOSE.

3. Вызовите SetFileInformationByHandle для открытого файла.

Мы создадим проект консольного приложения с именем DelTest, для которого текст использования должен быть что-то вроде этого:

c:\book>deltest

Usage: deltest.exe <method> <filename>

Method: 1=DeleteFile, 2=delete on close, 3=SetFileInformation.

Давайте рассмотрим код пользовательского режима для каждого из этих методов

Использование DeleteFile тривиально:

BOOL success = ::DeleteFile(filename);

Открыть файл с флагом удаления при закрытии можно следующим образом:

HANDLE hFile = ::CreateFile(filename, DELETE, 0, nullptr, OPEN_EXISTING,

FILE_FLAG_DELETE_ON_CLOSE, nullptr);

::CloseHandle(hFile);

При закрытии дескриптора, файл следует удалить (если драйвер не препятствует этому!)

Наконец, используя SetFileInformationByHandle:

```
FILE_DISPOSITION_INFO info;

info.DeleteFile = TRUE;

HANDLE hFile = ::CreateFile(filename, DELETE, 0, nullptr, OPEN_EXISTING, 0,
nullptr);

BOOL success = ::SetFileInformationByHandle(hFile, FileDispositionInfo,

            &info, sizeof(info));

::CloseHandle(hFile);
```

С помощью этого инструмента мы можем протестировать наш драйвер. Вот некоторые примеры:

C:\book>fltmc load delprotect2

C:\book>DelProtectConfig.exe add deltest.exe

Success.

C:\book>DelTest.exe

Usage: deltest.exe <method> <filename>

Method: 1=DeleteFile, 2=delete on close, 3=SetFileInformation.

C:\book>DelTest.exe 1 hello.txt

Using DeleteFile:

Error: 5

C:\book>DelTest.exe 2 hello.txt

Using CreateFile with FILE_FLAG_DELETE_ON_CLOSE:

Error: 5

C:\book>DelTest.exe 3 hello.txt

Using SetFileInformationByHandle:

Error: 5

C:\book>DelProtectConfig.exe remove deltest.exe

Success.

C:\book>DelTest.exe 1 hello.txt

Using DeleteFile:

Success!

Имена файлов

В некоторых обратных вызовах мини-фильтров в большинстве случаев требуется имя файла, к которому осуществляется доступ.

Поначалу кажется, что найти эту деталь достаточно легко: структура FILE_OBJECT имеет FileName.

К сожалению, не все так просто. Файлы могут открываться с полным или относительным путем.

Переименования одного и того же файла могут выполняться одновременно, некоторая информация об имени файла кешируется. По этим и другим внутренним причинам полю FileName в файловом объекте нельзя доверять.

Фактически, он гарантированно будет действителен только в обратном вызове перед операцией IRP_MJ_CREATE, и даже там это не обязательно в том формате, который нужен драйверу.

Чтобы решить эту проблему, диспетчер фильтров предоставляет API FltGetFileNameInformation, который может при необходимости вернуть правильное имя файла.

Прототип этой функции выглядит следующим образом:

```
NTSTATUS FltGetFileNameInformation (
            _In_ PFLT_CALLBACK_DATA CallbackData,
            _In_ FLT_FILE_NAME_OPTIONS NameOptions,
            _Outptr_ PFLT_FILE_NAME_INFORMATION *FileNameInformation);
```

Параметр CallbackData предоставляется диспетчером фильтров в любом обратном вызове.

Параметр NameOptions - это набор флагов, которые определяют (среди прочего) запрошенный формат файла.Типичное значение, используемое большинством драйверов - FLT_FILE_NAME_NORMALIZED (полное имя пути) ИЛИ с FLT_FILE_NAME_QUERY_DEFAULT (Ищет имя в кеше, в противном случае запрашивает файловую систему).

Результат вызова предоставляется последним параметром FileNameInformation. В результате выделяется структура, которую необходимо правильно освободить, вызвав FltReleaseFileNameInformation.

Структура FLT_FILE_NAME_INFORMATION определяется так:

```
typedef struct _FLT_FILE_NAME_INFORMATION {
      USHORT Size;
      FLT_FILE_NAME_PARSED_FLAGS NamesParsed;
      FLT_FILE_NAME_OPTIONS Format;
      UNICODE_STRING Name;
      UNICODE_STRING Volume;
      UNICODE_STRING Share;
      UNICODE_STRING Extension;
      UNICODE_STRING Stream;
      UNICODE_STRING FinalComponent;
      UNICODE_STRING ParentDir;
} FLT_FILE_NAME_INFORMATION, *PFLT_FILE_NAME_INFORMATION;
```

Основными параметрами являются несколько структур UNICODE_STRING, которые должны содержать различные компоненты имени файла. Первоначально только поле Name инициализируется полным именем файла (в зависимости от флагов, используемых для запроса информации об имени файла). Если в запросе указан флаг FLT_FILE_NAME_NORMALIZED, тогда Name указывает на полное имя пути в устройстве.

Форма устройства означает, что такой файл, как c:\mydir\myfile.txt, хранится с внутренним именем устройства, которому соответствует «C:», например \Device\HarddiskVolume3\mydir\myfile.txt.

Это делает драйвер немного сложнее, если он так или иначе зависит от путей, предоставленных пользовательским режимом (подробнее об этом позже).

Поскольку по умолчанию (поле "Имя") предоставляется только полное имя, часто бывает необходимо разделить полный путь к его составляющим. К счастью, менеджер фильтров предоставляет такую услугу с FltParseFileNameInformation.

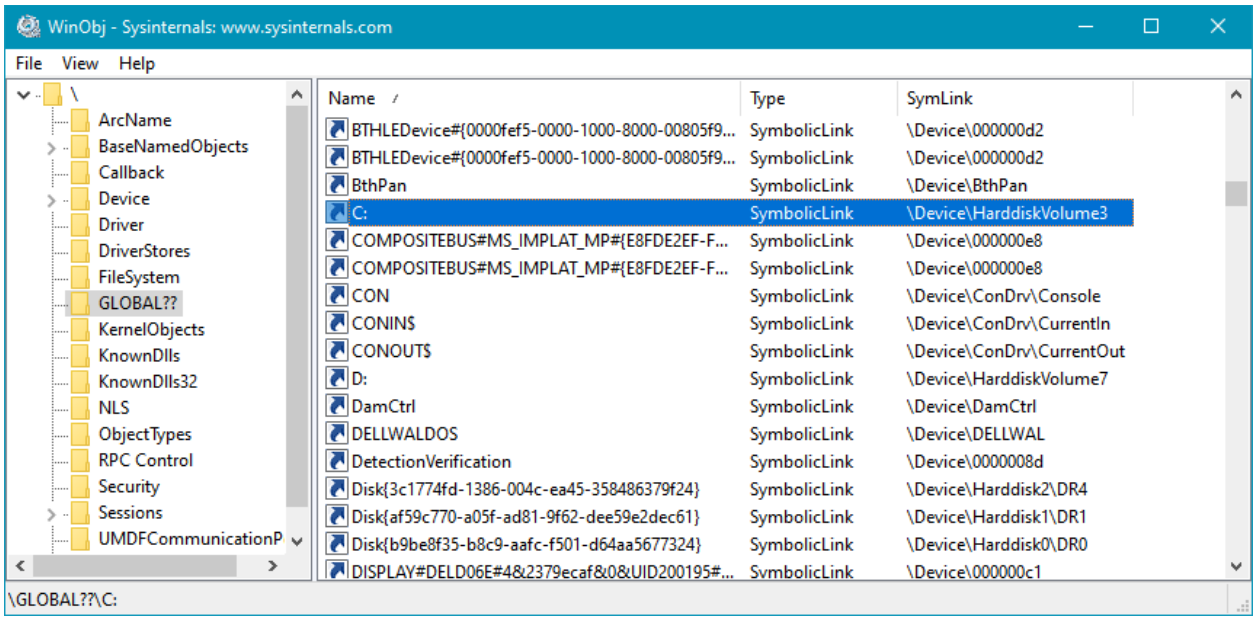
Она берет объект FLT_FILE_NAME_INFORMATION и заполняет остальные поля UNICODE_STRING в структуре.

Части имени файла

Как видно из объявления FLT_FILE_NAME_INFORMATION, есть несколько компонентов, которые составьте полное имя файла. Вот пример локального файла «c:\mydir1\mydir2\myfile.txt»:

Том - это фактическое имя устройства, которому соответствует символическая ссылка «C:».

На рисунке показано WinObj показывающий символическую ссылку C: и ее цель, которой является \Device\HarddiskVolume3 на этой машине.



Строка общего доступа пуста для локальных файлов (длина равна нулю).

ParentDir устанавливается только для каталога. В нашем примере это будет \mydir1\mydir2\ (а не обратная косая черта в конце).

Расширение - это просто расширение файла. В нашем примере это txt. В поле FinalComponent хранится имя файла и имя потока (если не используется поток по умолчанию). В нашем примере это myfile.txt.

Компонент Stream дает некоторые пояснения. Некоторые файловые системы (наиболее известная NTFS) предоставляют возможность иметь несколько «потоков» данных в одном файле. По сути, это означает, что несколько файлов могут хранятся в одном «физическом» файле.

В NTFS, например, то, что мы обычно считаем данными файла фактически является одним из его потоков с именем «$ DATA», который считается потоком по умолчанию.

Но это возможно для создания/открытия другого потока, который, так сказать, хранится в том же файле. Такие инструменты, как Windows Explorer не ищет эти потоки, и размеры любых альтернативных потоков не отображаются или возвращается стандартными API, такими как GetFileSize.

Имена потоков указываются с двоеточием после имени файла перед именем самого потока. Например, имя файла «myfile.txt: mystream» указывает на альтернативный поток с именем «mystream» в файле «myfile.txt».

Могут быть созданы альтернативные потоки с интерпретатором команд, как показано в следующем примере:

```
C:\temp>echo hello > hello.txt:mystream

C:\Temp>dir hello.txt
 Volume in drive C is OS
 Volume Serial Number is 1707-9837

 Directory of C:\Temp

22-May-19  11:33                0 hello.txt
               1 File(s)              0 bytes
```

Обратите внимание на нулевой размер файла. Данные действительно там?

Мы можем использовать инструмент SysInternals

Streams.exe, чтобы перечислить имена и размеры альтернативных потоков в файлах. Вот команда с нашим файлом hello.txt:

C:\Temp>streams -nobanner hello.txt

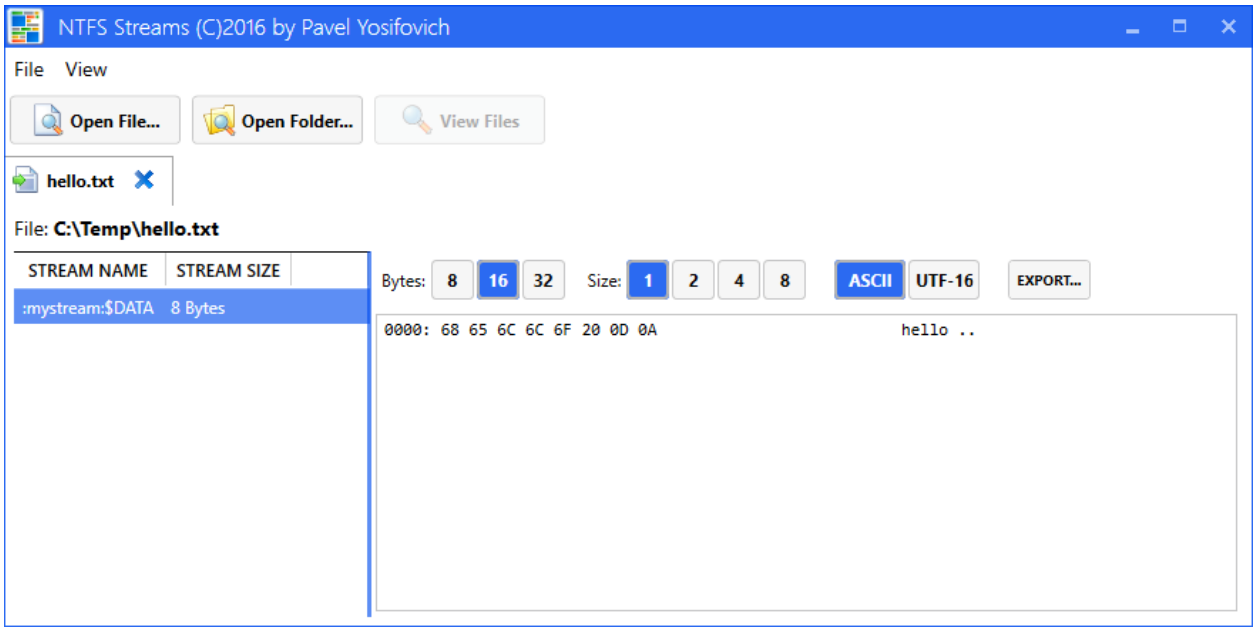
C:\Temp\hello.txt:

:mystream:$DATA 8

Альтернативное содержимое потока не отображается. Чтобы просмотреть (и при необходимости экспортировать в другой файл) stream данных, мы можем использовать инструмент под названием NtfsStreams.

На рисунке показано, как NtfsStreams открывает файл hello.txt из предыдущего примера.

Мы можем ясно посмотреть размер потока и данные.



Конечно, альтернативные потоки могут быть созданы программно, передав имя потока в конце имени файла после двоеточия в CreateFile API. Вот пример (обработка ошибок опущена):

```
HANDLE hFile = ::CreateFile(L"c:\\temp\\myfile.txt:stream1",
            GENERIC_WRITE, 0, nullptr, OPEN_ALWAYS, 0, nullptr);
char data[] = "Hello, from a stream";
DWORD bytes;
::WriteFile(hFile, data, sizeof(data), &bytes, nullptr);
::CloseHandle(hFile);
```

RAII FLT_FILE_NAME_INFORMATION wrapper

Как обсуждалось в предыдущем разделе, для вызова FltGetFileNameInformation необходимо вызвать функцию FltReleaseFileNameInformation. Это, естественно, приводит к возможности наличие обертки RAII, которая позаботится об этом, что сделает окружающий код более простым и меньшим количеством ошибок.

Вот одно возможное объявление такой оболочки:

```
enum class FileNameOptions {
      Normalized = FLT_FILE_NAME_NORMALIZED,
                  = FLT_FILE_NAME_OPENED,
                  Opened
                  Short
                  = FLT_FILE_NAME_SHORT,
                  = FLT_FILE_NAME_QUERY_DEFAULT,
```

```
                  QueryDefault

                  = FLT_FILE_NAME_QUERY_CACHE_ONLY,

                  QueryCacheOnly

                  QueryFileSystemOnly = FLT_FILE_NAME_QUERY_FILESYSTEM_ONLY,

                  RequestFromCurrentProvider =
FLT_FILE_NAME_REQUEST_FROM_CURRENT_PROVIDER,

                  DoNotCache

                  = FLT_FILE_NAME_DO_NOT_CACHE,

                  = FLT_FILE_NAME_ALLOW_QUERY_ON_REPARSE

                  AllowQueryOnReparse

};

DEFINE_ENUM_FLAG_OPERATORS(FileNameOptions);

struct FilterFileNameInformation {

      FilterFileNameInformation(PFLT_CALLBACK_DATA data, FileNameOptions
options =

                  FileNameOptions::QueryDefault |
FileNameOptions::Normalized);

      ~FilterFileNameInformation();

      operator bool() const {

            return _info != nullptr;

      }

      operator PFLT_FILE_NAME_INFORMATION() const {

            return Get();

      }

      PFLT_FILE_NAME_INFORMATION operator->() {

            return _info;

      }

      NTSTATUS Parse();

private:

      PFLT_FILE_NAME_INFORMATION _info;

};
```

Не встроенные функции определены ниже:

```
FilterFileNameInformation::FilterFileNameInformation(
            PFLT_CALLBACK_DATA data, FileNameOptions options) {
      auto status = FltGetFileNameInformation(data,
                  (FLT_FILE_NAME_OPTIONS)options, &_info);
      if (!NT_SUCCESS(status))
            _info = nullptr;
}
FilterFileNameInformation::~FilterFileNameInformation() {
      if (_info)
            FltReleaseFileNameInformation(_info);
}
NTSTATUS FilterFileNameInformation::Parse() {
      return FltParseFileNameInformation(_info);
}
```

Использование этой оболочки может быть примерно таким:

```
FilterFileNameInformation nameInfo(Data);
if(nameInfo) { // operator bool()
      if(NT_SUCCESS(nameInfo.Parse())) {
            KdPrint(("Final component: %wZ\n", &nameInfo->FinalComponent));
      }
}
```

Альтернативный драйвер защиты от удаления

Давайте создадим альтернативный драйвер Delete Protector, который будет защищать удаление файлов в определенных каталогах (независимо от вызывающего процесса).

Во-первых, нам нужно будет управлять защищаемыми каталогами (а не обрабатывать имена файлов, как в старом драйвере). Сложность возникает из-за того, что клиент пользовательского режима будет использовать каталоги в форме «C:\somedir»; то есть пути, основанные на символических ссылках.

Как мы уже видели, драйвер получает настоящее устройство, а не символические ссылки. Это означает, что нам нужно как-то преобразовать в стиле DOS имена (как их иногда называют) к именам в стиле NT (еще одно частое обращение к внутренним именам устройств).

С этой целью в нашем списке защищенных каталогов каждый каталог будет представлен в двух формах.

Вот определение структуры:

```
struct DirectoryEntry {
	UNICODE_STRING DosName;
	UNICODE_STRING NtName;
	void Free() {
		if (DosName.Buffer) {
			ExFreePool(DosName.Buffer);
			DosName.Buffer = nullptr;
		}
		if (NtName.Buffer) {
			ExFreePool(NtName.Buffer);
			NtName.Buffer = nullptr;
		}
	}
};
```

Поскольку мы будем распределять эти строки динамически, в конечном итоге нам нужно освободить их. Приведенный выше код добавляет метод Free, который освобождает внутренние строковые буферы.

Я решил использовать объекты UNICODE_STRING потому что сами строки могут быть размещены динамически, а некоторые API работают с UNICODE_STRING напрямую.

Теперь мы можем хранить массив этих структур и управлять им аналогично тому, как это было сделано в предыдущем драйвере:

```
const int MaxDirectories = 32;
DirectoryEntry DirNames[MaxDirectories];
int DirNamesCount;
FastMutex DirNamesLock;
```

Коды управления вводом/выводом предыдущего драйвера изменили значение - они должны позволять добавлять и удалять каталоги, которые, конечно же, также являются строками.

Вот обновленные определения:

```
#define IOCTL_DELPROTECT_ADD_DIR
```

```
\
CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DELPROTECT_REMOVE_DIR \
CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DELPROTECT_CLEAR
\
CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Затем нам нужно реализовать эти операции добавления/удаления/очистки, начиная с добавления.

Первая часть, делаем некоторые проверки для входной строки:

```
case IOCTL_DELPROTECT_ADD_DIR:
{
	auto name = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
	if (!name) {
		status = STATUS_INVALID_PARAMETER;
		break;
	}
	auto bufferLen = stack->Parameters.DeviceIoControl.InputBufferLength;
	if (bufferLen > 1024) {
		// just too long for a directory
		status = STATUS_INVALID_PARAMETER;
		break;
	}
	// make sure there is a NULL terminator somewhere
	name[bufferLen / sizeof(WCHAR) - 1] = L'\0';
	auto dosNameLen = ::wcslen(name);
	if (dosNameLen < 3) {
		status = STATUS_BUFFER_TOO_SMALL;
		break;
	}
```

Теперь, когда у нас есть правильный буфер, мы должны проверить, существует ли он уже в нашем массиве, и если нет нужно добавить его. Мы создадим вспомогательную функцию для этого поиска:

```
int FindDirectory(PCUNICODE_STRING name, bool dosName) {
	if (DirNamesCount == 0)
		return -1;
	for (int i = 0; i < MaxDirectories; i++) {
		const auto& dir = dosName ? DirNames[i].DosName :
DirNames[i].NtName;
		if (dir.Buffer && RtlEqualUnicodeString(name, &dir, TRUE))
			return i;
	}
	return -1;
}
```

Функция просматривает массив в поисках совпадения с входной строкой. Логический параметр указывает, сравнивается ли имя DOS или имя NT.
Мы используем RtlEqualUnicode - Функция для проверки равенства строк без учета регистра.
Функция **FindDirectory** возвращает индекс, в котором была найдена строка, или -1 в противном случае. Теперь наш обработчик добавления каталога может искать строку входного каталога и просто двигаться дальше, если он найден:

```
AutoLock locker(DirNamesLock);
UNICODE_STRING strName;
RtlInitUnicodeString(&strName, name);
if (FindDirectory(&strName, true) >= 0) {
	// found it, just continue and return success
	break;
}
```

Получив быстрый мьютекс, мы можем безопасно перейти к доступу массива каталогов. Если строка не найдена, у нас есть новый каталог для добавления в массив.

Во-первых, мы должны убедиться, что массив не переполнен:

```
if (DirNamesCount == MaxDirectories) {
	status = STATUS_TOO_MANY_NAMES;
	break;
}
```

На этом этапе нам нужно пройти по массиву и найти пустой слот (где буфер строки DOS — это указатель на NULL).

Как только мы его найдем, мы добавим имя DOS и сделаем что-нибудь, чтобы преобразовать его в NT имя, которое нам обязательно понадобится позже.

```
for (int i = 0; i < MaxDirectories; i++) {
	if (DirNames[i].DosName.Buffer == nullptr) {
		// allow space for trailing backslash and NULL terminator
		auto len = (dosNameLen + 2) * sizeof(WCHAR);
		auto buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool, len,
DRIVER_TAG);
		if (!buffer) {
			status = STATUS_INSUFFICIENT_RESOURCES;
			break;
		}
		::wcscpy_s(buffer, len / sizeof(WCHAR), name);
		// append a backslash if it's missing
		if (name[dosNameLen - 1] != L'\\')
			::wcscat_s(buffer, dosNameLen + 2, L"\\");
		if (!NT_SUCCESS(status)) {
			ExFreePool(buffer);
			break;
		}
		RtlInitUnicodeString(&DirNames[i].DosName, buffer);
		KdPrint(("Add: %wZ <=> %wZ\n", &DirNames[i].DosName,
&DirNames[i].NtName));
		++DirNamesCount;
		break;
	}
}
```

Код довольно прост, за исключением вызова ConvertDosNameToNtName. Этот код не встроенная функция, а то, что нам нужно реализовать самостоятельно.

Вот его прототип:

```
NTSTATUS ConvertDosNameToNtName(_In_ PCWSTR dosName, _Out_ PUNICODE_STRING ntName);
```

Как мы можем преобразовать имя DOS в имя NT ?

Подход будет заключаться в поиске символической ссылки и нахождении ее цели, которой является имя NT.

Мы начнем с некоторых базовых проверок:

```
ntName->Buffer = nullptr;

// in case of failure

auto dosNameLen = ::wcslen(dosName);

if (dosNameLen < 3)

      return STATUS_BUFFER_TOO_SMALL;

// make sure we have a driver letter

if (dosName[2] != L'\\' || dosName[1] != L':')

      return STATUS_INVALID_PARAMETER;
```

Мы ожидаем, что каталог будет иметь вид «X: \ ...», что означает букву диска, двоеточие, обратную косую черту и остальные символы - это путь. Мы не принимаем общие ресурсы в этом драйвере (например, «\myserver\myshare\mydir»).

Теперь нам нужно создать символическую ссылку, находящуюся в каталоге \??\ Object Manager.

Мы можем использовать функции манипулирования строками для создания полной строки.

В следующем фрагменте кода мы будем использовать тип под названием kstring, который представляет собой строковую оболочку, аналогичную по концепции стандартному C ++ std :: wstring.

Мы начнем с базового каталога символических ссылок и добавим предоставленную букву диска:

```
kstring symLink(L"\\??\\");

symLink.Append(dosName, 2);
```

Теперь нам нужно открыть символическую ссылку с помощью ZwOpenSymbolicLinkObject.

Для этого нам необходимо подготовить структуру OBJECT_ATTRIBUTES.

```
UNICODE_STRING symLinkFull;

symLink.GetUnicodeString(&symLinkFull);

OBJECT_ATTRIBUTES symLinkAttr;

InitializeObjectAttributes(&symLinkAttr, &symLinkFull,

            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);
```

GetUnicodeString - это вспомогательная функция kstring, которая инициализирует UNICODE_STRING на основе kstring.

Это необходимо, поскольку для имени OBJECT_ATTRIBUTES требуется UNICODE_STRING.

Иниинициализация OBJECT_ATTRIBUTES выполняется с помощью макроса InitializeObjectAttributes, требуя следующие аргументы (по порядку):

• Указатель на структуру OBJECT_ATTRIBUTES для инициализации.

• Название объекта.

• Набор флагов. В этом случае возвращаемый дескриптор будет дескриптором ядра, а поиск будет без учета регистра.

• Необязательный дескриптор корневого каталога в случае, если имя является относительным, а не абсолютным. NULLв нашем случае.

• Необязательный дескриптор безопасности для применения к объекту (если он создан и не открыт). NULL в нашем/

После инициализации структуры мы готовы вызывать ZwOpenSymbolicLinkObject:

```
HANDLE hSymLink = nullptr;

auto status = STATUS_SUCCESS;

do {

// open symbolic link

status = ZwOpenSymbolicLinkObject(&hSymLink, GENERIC_READ, &symLinkAttr);

if (!NT_SUCCESS(status))

break;
```

Мы будем использовать хорошо известную схему do/while (false) для очистки дескриптора.

ZwOpenSymbolicLinkObject принимает выходной HANDLE, маску доступа (GENERIC_READ здесь означает, что мы просто хотим прочитать информацию) и атрибуты, которые мы подготовили ранее.

Этот вызов, безусловно, может потерпеть неудачу, если например, указанная буква диска не существует.

Если вызов завершился успешно, нам нужно прочитать цель, на которую указывает этот объект символической ссылки.

Функция для получения этой информация - ZwQuerySymbolicLinkObject. Нам нужно подготовить UNICODE_STRING достаточно большого размера:

```
USHORT maxLen = 1024;
ntName->Buffer = (WCHAR*)ExAllocatePool(PagedPool, maxLen);
if (!ntName->Buffer) {
	status = STATUS_INSUFFICIENT_RESOURCES;
	break;
}
ntName->MaximumLength = maxLen;
// read target of symbolic link
status = ZwQuerySymbolicLinkObject(hSymLink, ntName, nullptr);
if (!NT_SUCCESS(status))
	break;
} while (false);
```

После выхода из блока do/while нам нужно освободить выделенный буфер, если что-то не удалось.

Иначе, мы можем добавить оставшуюся часть входного каталога к полученному целевому имени NT:

```
if (!NT_SUCCESS(status)) {
	if (ntName->Buffer) {
		ExFreePool(ntName->Buffer);
		ntName->Buffer = nullptr;
	}
}
else {
	RtlAppendUnicodeToString(ntName, dosName + 2);
}
```

Наконец, нам просто нужно закрыть дескриптор символической ссылки в случае успешного открытия:

```
if (hSymLink)
      ZwClose(hSymLink);
return status;
}
```

Для удаления защищенного каталога мы делаем аналогичные проверки по указанному пути, а затем смотрим его. Если найдено, удаляем его из массива:

```
AutoLock locker(DirNamesLock);
UNICODE_STRING strName;
RtlInitUnicodeString(&strName, name);
int found = FindDirectory(&strName, true);
if (found >= 0) {
      DirNames[found].Free();
      DirNamesCount--;
}
else {
      status = STATUS_NOT_FOUND;
}
break;
```

Обработка предварительно созданной и предварительно установленной информации

Теперь, когда у нас есть указанная выше инфраструктура, мы можем переключить наше внимание на реализацию предварительных обратных вызывов, чтобы любой файл в одном из защищенных каталогов не был удален, независимо от вызывающего процесса.

В обоих обратных вызовах нам нужно получить имя файла, который нужно удалить, и найти его каталог в нашем массиве каталогов.

Для этой цели мы создадим вспомогательную функцию, объявленную так:

bool IsDeleteAllowed(_In_ PFLT_CALLBACK_DATA Data);

Поскольку рассматриваемый файл спрятан в структуре FLT_CALLBACK_DATA, это все, что нам понадобиться. Первое, что нужно сделать, это получить имя файла:

```
PFLT_FILE_NAME_INFORMATION nameInfo = nullptr;

auto allow = true;

do {

      auto status = FltGetFileNameInformation(Data,

                  FLT_FILE_NAME_QUERY_DEFAULT | FLT_FILE_NAME_NORMALIZED,
&nameInfo);

      if (!NT_SUCCESS(status))

            break;

      status = FltParseFileNameInformation(nameInfo);

      if (!NT_SUCCESS(status))

            break;
```

Мы получаем информацию об имени файла и затем анализируем ее, так как нам нужен том и родительский каталог.

Нам нужно создать UNICODE_STRING, который объединяет эти три фактора:

```
// concatenate volume+share+directory

UNICODE_STRING path;

path.Length = path.MaximumLength =

            nameInfo->Volume.Length + nameInfo->Share.Length + nameInfo-
>ParentDir.Length;

path.Buffer = nameInfo->Volume.Buffer;
```

Поскольку полный путь к файлу в памяти является непрерывным, указатель буфера начинается с первого компонента и длина должна быть рассчитана соответствующим образом.

Все, что осталось сделать, это вызвать FindDirectory, чтобы найти (или не найти) этот каталог:

```
AutoLock locker(DirNamesLock);

if (FindDirectory(&path, false) >= 0) {

      allow = false;

      KdPrint(("File not allowed to delete: %wZ\n", &nameInfo->Name));

}

} while (false);
```

Наконец, освободите информацию об имени файла, и все готово:

```
if (nameInfo)
	FltReleaseFileNameInformation(nameInfo);
return allow;
}
```

Вернемся к нашим предварительным обратным вызовам.

```
_Use_decl_annotations_
FLT_PREOP_CALLBACK_STATUS DelProtectPreCreate(PFLT_CALLBACK_DATA Data,
		PCFLT_RELATED_OBJECTS, PVOID*) {
	if (Data->RequestorMode == KernelMode)
		return FLT_PREOP_SUCCESS_NO_CALLBACK;
	auto& params = Data->Iopb->Parameters.Create;
	if (params.Options & FILE_DELETE_ON_CLOSE) {
		// delete operation
		KdPrint(("Delete on close: %wZ\n", &FltObjects->FileObject-
>FileName));
		if (!IsDeleteAllowed(Data)) {
			Data->IoStatus.Status = STATUS_ACCESS_DENIED;
			return FLT_PREOP_COMPLETE;
		}
	}
	return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

И DelProtectPreSetInformation, которая очень похожа:

```
_Use_decl_annotations_

FLT_PREOP_CALLBACK_STATUS DelProtectPreSetInformation(PFLT_CALLBACK_DATA
Data,

            PCFLT_RELATED_OBJECTS, PVOID*) {

      if (Data->RequestorMode == KernelMode)

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      auto& params = Data->Iopb->Parameters.SetFileInformation;

      if (params.FileInformationClass != FileDispositionInformation &&

                  params.FileInformationClass !=
FileDispositionInformationEx) {

            // not a delete operation

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      }

      auto info = (FILE_DISPOSITION_INFORMATION*)params.InfoBuffer;

      if (!info->DeleteFile)

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      if (IsDeleteAllowed(Data))

            return FLT_PREOP_SUCCESS_NO_CALLBACK;

      Data->IoStatus.Status = STATUS_ACCESS_DENIED;

      return FLT_PREOP_COMPLETE;

}
```

Тестирование драйвера

Клиент конфигурации был обновлен для отправки обновленных управляющих кодов (проект названный DelProtectConfig3 в источниках к книге). Вот несколько тестов:

```
c:\book>fltmc load delprotect3

c:\book>delprotectconfig3 add c:\users\pavel\pictures

Success!

c:\book>del c:\users\pavel\pictures\pic1.jpg

c:\users\pavel\pictures\pic1.jpg

Access is denied.
```

Контексты

В некоторых ситуациях желательно прикрепить некоторые данные к объектам файловой системы, таким как тома и файлы.

Диспетчер фильтров предоставляет эту возможность через контексты.

Контекст - это предоставленная структура данных драйвером мини-фильтра, который может быть установлен и получен для любого объекта файловой системы.

Эти контексты связаны с объектами, на которых они установлены, пока эти объекты живы.

Чтобы использовать контексты, драйвер должен заранее объявить, какие контексты ему могут потребоваться и для какого типа объектов.

Это делается как часть структуры регистрации FLT_REGISTRATION.

Поле ContextRegistration может указывать на массив структур FLT_CONTEXT_REGISTRATION, каждый из которых определяет информацию для одного контекста.

FLT_CONTEXT_REGISTRATION объявлен так:

```
typedef struct _FLT_CONTEXT_REGISTRATION {
      FLT_CONTEXT_TYPE ContextType;
      FLT_CONTEXT_REGISTRATION_FLAGS Flags;
      PFLT_CONTEXT_CLEANUP_CALLBACK ContextCleanupCallback;
      SIZE_T Size;
      ULONG PoolTag;
      PFLT_CONTEXT_ALLOCATE_CALLBACK ContextAllocateCallback;
      PFLT_CONTEXT_FREE_CALLBACK ContextFreeCallback;
      PVOID Reserved1;
} FLT_CONTEXT_REGISTRATION, *PFLT_CONTEXT_REGISTRATION;
```

Вот описание вышеуказанных полей:

• ContextType определяет тип объекта, к которому будет прикреплен этот контекст. FLT_CONTEXT_TYPE определяется как USHORT и может принимать одно из следующих значений:

```
    #define FLT_VOLUME_CONTEXT          0x0001
    #define FLT_INSTANCE_CONTEXT        0x0002
    #define FLT_FILE_CONTEXT            0x0004
    #define FLT_STREAM_CONTEXT          0x0008
    #define FLT_STREAMHANDLE_CONTEXT    0x0010
    #define FLT_TRANSACTION_CONTEXT     0x0020
#if FLT_MGR_WIN8
    #define FLT_SECTION_CONTEXT         0x0040
#endif // FLT_MGR_WIN8
    #define FLT_CONTEXT_END             0xffff
```

Как видно из приведенных выше определений, контекст может быть прикреплен к тому, экземпляру фильтра, файлу, потоку, дескриптору потока, транзакции и разделу (в Windows 8 и новее). Последнее значение - для указания, что это конец списка определений контекста.

Размер контекста может быть фиксированным или переменным. Если требуется фиксированный размер, он указывается в поле «Size» FLT_CONTEXT_REGISTRATION.

Для контекста переменного размера драйвер указывает специальное значение FLT_VARIABLE_SIZED_CONTEXTS (-1). Использование контекстов фиксированного размера более эффективно, потому что диспетчер фильтров может использовать дополнительные списки для управления выделением и освобождением (см. WDK документацию для получения дополнительных сведений о дополнительных списках).

Тег пула указывается в поле PoolTag FLT_CONTEXT_REGISTRATION.

Это тег диспетчера фильтров будет использовать при фактическом распределении контекста.

Следующие два поля необязательны, обратные вызовы, в которых драйвер предоставляет функции выделения и освобождения. Если они не равны NULL, тогда поля PoolTag и Size не имеют смысла и не используются.

Вот пример построения массива структуры регистрации контекста:

```
struct FileContext {

      //...

};

const FLT_CONTEXT_REGISTRATION ContextRegistration[] = {

            { FLT_FILE_CONTEXT, 0, nullptr, sizeof(FileContext), 'torP',

                        nullptr, nullptr, nullptr },

                        { FLT_CONTEXT_END }

};
```

Управление контекстами

Чтобы действительно использовать контекст, драйвер сначала должен выделить его, вызвав FltAllocateContext, который определен вот так:

```
NTSTATUS FltAllocateContext (
            _In_ PFLT_FILTER Filter,
            _In_ FLT_CONTEXT_TYPE ContextType,
            _In_ SIZE_T ContextSize,
            _In_ POOL_TYPE PoolType,
            _Outptr_ PFLT_CONTEXT *ReturnedContext);
```

Параметр Filter - это указатель фильтра, возвращаемый FltRegisterFilter, но также доступен в структуре FLT_RELATED_OBJECTS, предоставленной для всех обратных вызовов.

ContextType - один из поддерживаемых макросов контекста, показанные ранее, такие как FLT_FILE_CONTEXT.

ContextSize - это запрошенный размер контекста в байтах (должен быть больше нуля).

PoolType может иметь значение PagedPool или NonPagedPool, в зависимости от того, какой IRQL драйвер планирует получить доступ к контексту (для контекстов тома, необходимо указать NonPagedPool).

Наконец, поле ReturnedContext хранит возвращенные выделенные контексты; PFLT_CONTEXT, определяется как PVOID.

После выделения контекста драйвер может хранить в этом буфере данных все, что пожелает.

Затем он должен прикрепить контекст к объекту (это причина для создания контекста в первую очередь) используя одну из нескольких функций с именем FltSetXxxContext, где «Xxx» - это один из файлов, экземпляров, потоков, StreamHandle или транзакций.

Единственное исключение - контекст раздела, который устанавливается с помощью FltCreateSectionForDataScan.

Каждая из функций FltSetXxxContext имеет одинаковый общий вид:

```
NTSTATUS FltSetFileContext (
            _In_ PFLT_INSTANCE Instance,
            _In_ PFILE_OBJECT FileObject,
            _In_ FLT_SET_CONTEXT_OPERATION Operation,
            _In_ PFLT_CONTEXT NewContext,
            _Outptr_ PFLT_CONTEXT *OldContext);
```

Функция принимает необходимые параметры для текущего контекста. В данном случае это экземпляр (на самом деле необходимо в любой функции заданного контекста) и файловый объект, представляющий файл, который должен содержать этот контекст.

Параметр Operation может быть FLT_SET_CONTEXT_REPLACE_IF_EXISTS или FLT_SET_CONTEXT_KEEP_IF_EXISTS, которые не требуют пояснений.

NewContext - это контекст, который необходимо установить, а OldContext - это необязательный параметр, который можно использовать для получения предыдущего контекста с операцией, установленной на FLT_SET_CONTEXT_REPLACE_IF_EXISTS.

Контексты подсчитываются по ссылкам. Выделение контекста (FltAllocateContext) и установка контекста увеличивает счетчик ссылок. Противоположная функция - FltReleaseContext, которую нужно вызывать соответствующее количество раз, чтобы убедиться, что контекст будет удален.

Хотя есть функция удалить контекст (FltDeleteContext), обычно в этом нет необходимости, так как диспетчер фильтров удаляет контекст после уничтожения объекта файловой системы, в котором он находится.

Типичный сценарий - выделить контекст, заполнить его, установить для соответствующего объекта и затем вызвать FltReleaseContext один раз, сохраняя счетчик ссылок один для контекста.

Мы увидим практическое использование контекстов в разделе «Драйвер резервного копирования файлов» далее в этой главе.

После того, как для объекта был установлен контекст, другие обратные вызовы могут захотеть получить этот контекст.

Множество функций «get» обеспечивают доступ к соответствующему контексту, все они имеют имя в форме FltGetXxxContext, где «Xxx» - это один из файлов, экземпляров, томов, потоков, StreamHandle, транзакций или разделов.

Функции get увеличивают счетчик ссылок контекста, поэтому вызов FltReleaseContext необходимо после завершения работы с контекстом.

Инициирование запросов ввода-вывода

Мини-фильтры файловой системы иногда должны инициировать собственные операции ввода-вывода.

Обычно код ядра будет использовать такие функции, как ZwCreateFile, чтобы открыть дескриптор файла, а затем выполнить операции ввода-вывода с такими функциями, как ZwReadFile, ZwWriteFile, ZwDeviceIoControlFile и некоторыми другими.

Мини-фильтры обычно не используют ZwCreateFile, если им нужно выполнить операцию ввода-вывода из одного обратного вызова диспетчера фильтров.

Причина связана с тем, что операция ввода-вывода будет перемещаться от самого верхнего фильтра вниз к самой файловой системе, встречая текущий мини-фильтр на пути !

Это форма повторного входа, которая может вызвать проблемы, если драйвер не будет осторожен.

Это также снижает производительность из-за необходимости обхода всего стека фильтров файловой системы.

Вместо этого мини-фильтры используют процедуры диспетчера фильтров для выполнения операций ввода-вывода, которые отправляются к следующему нижнему фильтру по отношению к файловой системе, предотвращая повторный вход и возможное снижение производительности.

Эти API-интерфейсы начинаются с «Flt» и по концепции аналогичны вариантам «Zw».

Основная используемая функция:

FltCreateFile (и его расширенные друзья, FltCreateFileEx и FltCreateFileEx2.

Вот прототип FltCreateFile:

```
NTSTATUS FltCreateFile (
            _In_ PFLT_FILTER Filter,
            _In_opt_ PFLT_INSTANCE Instance,
            _Out_ PHANDLE FileHandle,
            _In_ ACCESS_MASK DesiredAccess,
            _In_ POBJECT_ATTRIBUTES ObjectAttributes,
            _Out_ PIO_STATUS_BLOCK IoStatusBlock,
            _In_opt_ PLARGE_INTEGER AllocationSize,
            _In_ ULONG FileAttributes,
            _In_ ULONG ShareAccess,
            _In_ ULONG CreateDisposition,
            _In_ ULONG CreateOptions,
            _In_reads_bytes_opt_(EaLength) PVOID EaBuffer,
            _In_ ULONG EaLength,
            _In_ ULONG Flags);
```

Ничего себе, у этой функции много-много вариантов. К счастью, это не так трудно понять, но они должны быть установлены правильно, иначе вызов не будет выполнен с каким-то странным статусом.

Как видно из объявления, первый аргумент - это адрес фильтра, используемый как базовый.

Основное возвращаемое значение — FileHandle, это указатель на открытый файл в случае успеха. Мы не будем рассматривать все различные параметры (см. WDK документацию), но мы будем использовать эту функцию в следующем разделе.)

С возвращенным дескриптором драйвер может вызывать стандартные API ввода-вывода, такие как ZwReadFile, ZwWriteFile и т. д.

Операция по-прежнему будет нацелена только на нижние уровни. В качестве альтернативы драйвер может использовать FILE_OBJECT из FltCreateFileEx или FltCreateFileEx2 с такими функциями, как FltReadFile и FltWriteFile (последние функции требуют файлового объекта, а не дескриптора).

После выполнения операции для возвращенного дескриптора необходимо вызвать FltClose.

Драйвер резервного копирования файлов

Пришло время применить полученные знания на практике, в частности, используя контексты и операции ввода-вывода в драйвере мини-фильтра.

Созданный нами драйвер обеспечивает автоматическое резервное копирование файла всякий раз, когда файл открывается для записи непосредственно перед записью.

Таким образом, можно вернуться к предыдущему состоянию файла при желании.

Фактически - у нас есть единственная резервная копия файла в любой момент.

Главный вопрос: где будет храниться эта резервная копия? Можно создать «резервную копию» в каталоге файла или, возможно, создать корневой каталог для всех резервных копий и заново создать резервную копию в той же структуре папок, начиная с исходного файла (драйвер может даже скрыть этот каталог от общего доступа).

Эти варианты хорошо, но для этой демонстрации мы воспользуемся другим вариантом:

Мы сохраним резервную копию файла в самом файле, в альтернативном потоке NTFS.

По сути, файл будет содержать собственную резервную копию.

Затем при необходимости мы можем поменять контексты альтернативного потока на поток по умолчанию, эффективно восстанавливая файл в предыдущее состояние.

Мы начнем с шаблона проекта мини-фильтра файловой системы, который мы использовали в предыдущих драйверах.

Имя драйвера будет FileBackup. Далее нам нужно исправить файл INF как обычно.

Вот детали которые были изменены:

```
[Version]
Signature   = "$Windows NT$"
Class       = "OpenFileBackup"
ClassGuid   = {f8ecafa6-66d1-41a5-899b-66585d7216b7}
Provider    = %ManufacturerName%
DriverVer   =
CatalogFile = FileBackup.cat

[MiniFilter.Service]
; truncated
LoadOrderGroup = "FS Open file backup filters"

[Strings]
; truncated
Instance1.Altitude    = "100200"
Instance1.Flags       = 0x0
```

Мы переименуем файл FileBackup.c в FileBackup.cpp для поддержки кода C ++.

Поскольку мы будем использовать альтернативные потоки, можно использовать только NTFS, поскольку это единственная «стандартная» файловая система в Windows для поддержки альтернативных файловых потоков.

Это означает, что драйвер не должен подключаться к тому, который не использует NTFS.

Драйверу необходимо изменить стандартную реализацию «настройки экземпляра».

Обратный вызов уже настроен шаблоном проекта. Вот полный код функции с дополнительной проверкой NTFS:

```
NTSTATUS FileBackupInstanceSetup(
            _In_ PCFLT_RELATED_OBJECTS FltObjects,
            _In_ FLT_INSTANCE_SETUP_FLAGS Flags,
            _In_ DEVICE_TYPE VolumeDeviceType,
            _In_ FLT_FILESYSTEM_TYPE VolumeFilesystemType) {
      UNREFERENCED_PARAMETER(FltObjects);
      UNREFERENCED_PARAMETER(Flags);
      UNREFERENCED_PARAMETER(VolumeDeviceType);
      if (VolumeFilesystemType != FLT_FSTYPE_NTFS) {
            KdPrint(("Not attaching to non-NTFS volume\n"));
            return STATUS_FLT_DO_NOT_ATTACH;
```

```
	}

	return STATUS_SUCCESS;

}
```

Возвращение STATUS_FLT_DO_NOT_ATTACH запрещает прикрепление к рассматриваемому тому.

Далее нам нужно зарегистрироваться на соответствующие запросы. Драйвер должен перехватывать операции записи,поэтому необходим обратный вызов перед операцией для IRP_MJ_WRITE.

Кроме того, нам нужно будет отслеживать некоторые состояния с использованием файлового контекста. Драйверу может потребоваться обработка операции после создания, а также очистка операция (IRP_MJ_CLEANUP).

Позже мы увидим, почему это необходимо. На данный момент, учитывая эти ограничения, мы можем настроить структуру регистрации обратного вызова следующим образом:

```
#define DRIVER_CONTEXT_TAG 'xcbF'

#define DRIVER_TAG 'bF'

const FLT_OPERATION_REGISTRATION Callbacks[] = {

		{ IRP_MJ_CREATE, 0, nullptr, FileBackupPostCreate },

		{ IRP_MJ_WRITE, FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO,

				FileBackupPreWrite, nullptr },

				{ IRP_MJ_CLEANUP, 0, nullptr,
FileBackupPostCleanup },

				{ IRP_MJ_OPERATION_END }

};
```

Затем нам понадобится некоторый контекст, чтобы отслеживать, выполнялась ли уже операция записи на конкретный открытый файл. Давайте определим структуру контекста, которую мы будем использовать:

```
struct FileContext {

	Mutex Lock;

	UNICODE_STRING FileName;

	BOOLEAN Written;

};
```

Мы сохраним само имя файла (его будет проще сохранить при резервном копировании файла), мьютекс для цели синхронизации и логическое значение, указывающее, выполнялась ли уже операция резервного копирования для этого файла.

Опять же, фактическое использование этого контекста станет более ясным, когда мы начнем обратный вызов.

Поскольку у нас есть контекст, его необходимо зарегистрировать в таком массиве контекстов:

```
const FLT_CONTEXT_REGISTRATION Contexts[] = {

            { FLT_FILE_CONTEXT, 0, nullptr, sizeof(FileContext),
DRIVER_CONTEXT_TAG },

            { FLT_CONTEXT_END }

};
```

На этот массив указывает полная структура регистрации, показанная ниже:

```
CONST FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),         //  Size
    FLT_REGISTRATION_VERSION,         //  Version
    0,                                //  Flags

    Contexts,                         //  Context
    Callbacks,                        //  Operation callbacks

    FileBackupUnload,                 //  MiniFilterUnload
    FileBackupInstanceSetup,
    FileBackupInstanceQueryTeardown,
    FileBackupInstanceTeardownStart,
    FileBackupInstanceTeardownComplete,
};
```

Теперь, когда у нас настроены все структуры, мы можем перейти к реализации обратного вызова.

Post Create Callback

Зачем нам вообще нужен обратный вызов для создания файла ?

Фактически можно написать драйвер и без него, но это поможет продемонстрировать некоторые функции, которых мы раньше не видели.

Наша цель — выделить контекст файла для интересующих нас файлов. Например, файлы, которые не открыты для записи, не имеют никакого интереса для драйвера.

Почему мы используем post-callback, а не pre-callback ?

Нам нужно, чтобы только если файл открывается успешно, то нашему драйверу следует изучать файл дальше.

Реализация начинается здесь:

```
FLT_POSTOP_CALLBACK_STATUS FileBackupPostCreate(
            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
            PVOID CompletionContext, FLT_POST_OPERATION_FLAGS Flags) {
```

Затем извлечем параметры операции создания:

```
const auto& params = Data→Iopb→Parameters.Create;
```

Нас интересуют только файлы, открытые для записи, а не из режима ядра, и не новые файлы (поскольку новые файлы не требуют резервного копирования). Вот что нужно сделать:

```
if (Data->RequestorMode == KernelMode
            || (params.SecurityContext->DesiredAccess & FILE_WRITE_DATA) == 0
            || Data->IoStatus.Information == FILE_DOES_NOT_EXIST) {
      // kernel caller, not write access or a new file - skip
      return FLT_POSTOP_FINISHED_PROCESSING;
}
```

Подобные проверки важны, так как они устраняют для драйвера много возможных накладных расходов.

Драйвер всегда должен стремиться делать как можно меньше, чтобы снизить влияние на производительность.

Теперь, когда у нас есть файл, который нам нужен, нам нужно подготовить объект контекста, который будет прикреплен к файлу.

Этот контекст понадобится позже, когда мы будем обрабатывать обратный вызов перед записью. Сначала извлечем имя файла.

Драйвер должен вызвать стандартный FltGetFileNameInformation.

Чтобы сделать это немного проще и менее подвержено ошибкам, мы будем использовать оболочку RAII, представленную ранее в этой главе:

```
FilterFileNameInformation fileNameInfo(Data);
if (!fileNameInfo) {
	return FLT_POSTOP_FINISHED_PROCESSING;
}
if (!NT_SUCCESS(fileNameInfo.Parse())) // FltParseFileNameInformation
	return FLT_POSTOP_FINISHED_PROCESSING;
```

Следующий шаг - решить, будем ли мы делать резервную копию всех файлов или файлов, находящихся в определенных каталогах.

Для гибкости, мы будем придерживаться второго подхода.

Давайте создадим вспомогательную функцию с именем IsBackupDirectory, которая должна возвращать истину для каталогов, которые нам небезразличны.

Вот простая реализация, которая возвращает true для любого каталога с именем «\ pictures \» или «\ documents \»:

```
bool IsBackupDirectory(_In_ PCUNICODE_STRING directory) {
	// no counted version of wcsstr :(
	ULONG maxSize = 1024;
	if (directory->Length > maxSize)
		return false;
	auto copy = (WCHAR*)ExAllocatePoolWithTag(PagedPool, maxSize + sizeof(WCHAR),
			DRIVER_TAG);
	if (!copy)
		return false;
	RtlZeroMemory(copy, maxSize + sizeof(WCHAR));
	wcsncpy_s(copy, 1 + maxSize / sizeof(WCHAR), directory->Buffer,
			directory->Length / sizeof(WCHAR));
	_wcslwr(copy);
	bool doBackup = wcsstr(copy, L"\\pictures\\") || wcsstr(copy, L"\\documents\\");
	ExFreePool(copy);
	return doBackup;
}
```

Функция принимает только имя каталога (извлеченное с помощью FltParseFileNameInformation) и необходимо искать указанные выше подстроки.

К сожалению, это не так просто, как хотелось бы.

Есть функция wcsstr, которая просматривает строку в поисках подстроки, но у нее есть две проблемы:

• Она чувствителен к регистру, что неудобно, когда речь идет о файлах или каталогах

• Она ожидает, что искомая строка будет завершена NULL, что не обязательно в случае с UNICODE_STRING.

Из-за вышеуказанных проблем код выделяет свой собственный строковый буфер, копирует имя каталога и преобразует строку в нижний регистр (_wcslwr) перед использованием wcsstr для поиска «\pictures\» и «\documents\».

Вернувшись к обратному вызову после создания, мы вызываем IsBackupDirectory и выходим, если он возвращает false:

```
if (!IsBackupDirectory(&fileNameInfo->ParentDir))
      return FLT_POSTOP_FINISHED_PROCESSING;
```

Далее нам нужно сделать еще одну проверку. Если открытый файл предназначен для потока не по умолчанию, тогда мы должны не создавать резервную копию чего-либо.

Мы сделаем резервную копию только потока данных по умолчанию:

```
if (fileNameInfo->Stream.Length > 0)
      return FLT_POSTOP_FINISHED_PROCESSING;
```

Наконец, мы готовы выделить контекст нашего файла и инициализировать его:

```
FileContext* context;
auto status = FltAllocateContext(FltObjects->Filter, FLT_FILE_CONTEXT,
            sizeof(FileContext), PagedPool, (PFLT_CONTEXT*)&context);
if (!NT_SUCCESS(status)) {
      KdPrint(("Failed to allocate file context (0x%08X)\n", status));
      return FLT_POSTOP_FINISHED_PROCESSING;
}
context->Written = FALSE;
context->FileName.MaximumLength = fileNameInfo->Name.Length;
context->FileName.Buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool,
            fileNameInfo->Name.Length, DRIVER_TAG);
```

```
if (!context->FileName.Buffer) {
      FltReleaseContext(context);
      return FLT_POSTOP_FINISHED_PROCESSING;
}
RtlCopyUnicodeString(&context->FileName, &fileNameInfo->Name);
// initialize mutex
context->Lock.Init();
```

Этот код требует объяснения. FltAllocateContext выделяет контекст с требуемым size и возвращает указатель на выделенную память.

Возвращенная контекстная память не обнуляется, поэтому все члены должны быть правильно инициализирован.

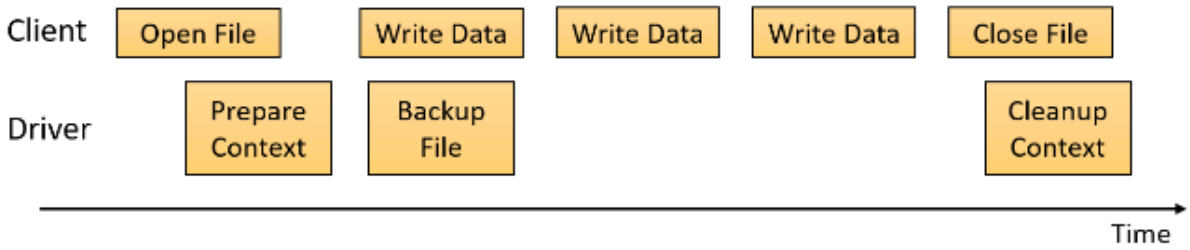
Зачем вообще нам нужен этот контекст? Типичный клиент открывает файл для записи, а затем вызывает WriteFile потенциально несколько раз.

Перед первым вызовом WriteFile драйвер должен вернуть предыдущее содержимое файла.

Вот почему нам нужно логическое поле Written - чтобы мы сделали резервную копию только один раз перед первой операцией записи.

Этот флаг начинается как False и меняется на true после первой операции записи.

Такой поворот событий изображен на следующем рисунке.



Затем мы выделяем память для хранения полного имени файла, которое нам понадобится позже при резервном копировании.

Технически мы можем вызвать FltGetFileNameInformation во время резервного копирования файла, но поскольку эта функция может не работать в некоторых ситуациях, лучше получить имя файла сейчас и использовать его позже, сделав драйвер более надежным.

Последнее поле в нашем контексте - мьютекс. Нам нужна некоторая синхронизация в необычном, на случай, когда более одного потока в клиентском процессе записывают в один и тот же файл примерно в в то же время. В таком случае нам нужно убедиться, что мы сделали единственную резервную копию данных, иначе наша резервная копия может быть повреждена.

Во всех до сих пор примерах, где нам требовалась такая синхронизация, мы использовали быстрый мьютекс, но здесь мы используем стандартный мьютекс.

Почему? Причина связана с операцией, которую драйвер будет вызывать при резервном копировании файла - API ввода-вывода, такие как ZwWriteFile и ZwReadFile может быть вызван только при IRQL PASSIVE_LEVEL (0).

Быстрый мьютекс поднимает IRQL до APC_LEVEL (1), что приведет к тупиковой ситуации, если используются API ввода-вывода.

Класс Mutex - тот же, что показан в главе 6, который будет использоваться с RAII AutoLock.

Теперь контекст инициализирован, поэтому нам нужно прикрепить его к файлу с FltSetFileContext и вернуться из обратного вызова post-create:

```
status = FltSetFileContext(FltObjects->Instance, FltObjects->FileObject,
            FLT_SET_CONTEXT_KEEP_IF_EXISTS, context, nullptr);
if (!NT_SUCCESS(status)) {
      KdPrint(("Failed to set file context (0x%08X)\n", status));
      ExFreePool(context->FileName.Buffer);
}
FltReleaseContext(context);
return FLT_POSTOP_FINISHED_PROCESSING;
}
```

Установленные контекстные API позволяют сохранить существующий контекст или заменить его (если есть).

Если контекст уже был возвращенный статус - STATUS_FLT_CONTEXT_ALREADY_DEFINED, что является статусом ошибки, и если возвращается статус ошибки, драйвер старается освободить ранее выделенный строковый буфер.

Наконец, необходимо вызвать FltReleaseContext, который, если все в порядке, устанавливает счетчик внутренних ссылок контекста в 1 (+1 для выделения, -1 для освобождения). Если контекст не удалось установить, он будет полностью освобожденным.

Обратный вызов перед записью

Задача обратного вызова перед записью - сделать копию данных файла непосредственно перед фактической операции записи.

Вот почему здесь необходим предварительный обратный вызов, иначе в пост-обратном вызове, операция уже будет завершена.

Начнем с получения контекста файла. Если он не существует, то мы можем просто двигаться дальше:

```
FLT_PREOP_CALLBACK_STATUS FileBackupPreWrite(
            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,
            PVOID* CompletionContext) {
      UNREFERENCED_PARAMETER(CompletionContext);
      UNREFERENCED_PARAMETER(Data);
      // get the file context if exists
      FileContext* context;
      auto status = FltGetFileContext(FltObjects->Instance,
                  FltObjects->FileObject, (PFLT_CONTEXT*)&context);
      if (!NT_SUCCESS(status) || context == nullptr) {
            // no context, continue normally
            return FLT_PREOP_SUCCESS_NO_CALLBACK;
      }
```

Когда у нас есть контекст, нам нужно сделать копию данных файла только один раз перед первой записью.

Сначала мы получаем мьютекс и проверяем записанный флаг из контекста, если это ложь, то резервная копия еще не создана, и мы вызываем вспомогательную функцию для создания резервной копии:

```
{
      AutoLock<Mutex> locker(context->Lock);
      if (!context->Written) {
            status = BackupFile(&context->FileName, FltObjects);
            if (!NT_SUCCESS(status)) {
                  KdPrint(("Failed to backup file! (0x%X)\n", status));
            }
```

```
            context->Written = TRUE;
        }
}
FltReleaseContext(context);
return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

Вспомогательная функция BackupFile - ключ к тому, чтобы все это работало. Можно подумать, что создание копирование файлов - это просто API; к сожалению, это не так.

В ядре нет функции «CopyFile».

API пользовательского режима CopyFile - это нетривиальная функция, которая выполняет довольно много работы по созданию копии.

Частично это чтение байтов из исходного файла и запись в целевой файл.

Но это не так. Во-первых, может быть несколько потоков для копирования (в случае NTFS).

Во-вторых, есть вопрос о дескрипторе безопасности из исходного файла, который также необходимо скопировать в определенных случаях (см. документацию для CopyFile, чтобы получить все подробности).

Суть в том, что нам нужно создать собственную операцию копирования файлов. К счастью, нам просто нужно скопировать один файловый поток - поток по умолчанию в другой поток внутри того же физического файла, что и наш резервный поток.

Вот начало нашей функции BackupFile:

```
NTSTATUS
BackupFile(_In_ PUNICODE_STRING FileName, _In_ PCFLT_RELATED_OBJECTS
FltObjects) {
        HANDLE hTargetFile = nullptr;
        HANDLE hSourceFile = nullptr;
        IO_STATUS_BLOCK ioStatus;
        auto status = STATUS_SUCCESS;
        void* buffer = nullptr;
```

Путь, который мы выберем - открыть два дескриптора - один (исходный) дескриптор, указывающий на исходный файл (с потоком по умолчанию для резервного копирования), а другой (целевой) дескриптор для резервного потока.

Затем мы прочитаем из источника и запишем в резервный поток. Концептуально это просто, но, как это часто бывает в ядре, дьявол кроется в деталях.)

Начнем с определения размера файла. Размер файла может быть нулевым, и в этом случае нечего копировать не нужно:

```
LARGE_INTEGER fileSize;

status = FsRtlGetFileSize(FltObjects->FileObject, &fileSize);

if (!NT_SUCCESS(status) || fileSize.QuadPart == 0)

      return status;
```

FsRtlGetFileSize рекомендуется всякий раз, когда требуется размер файла с учетом указателя на FILE_OBJECT.

Альтернативой было бы использование ZwQueryInformationFile для получения размера файла (у него многие другие типы информации, которые он может получить), но для этого требуется дескриптор файла, что в некоторых случаях может вызвать проблему.

Теперь мы готовы открыть исходный файл с помощью FltCreateFile. Важно не использовать ZwCreateFile, чтобы запросы ввода-вывода не отправлялись драйверу ниже нашего драйвера.

```
do {

      // open source file

      OBJECT_ATTRIBUTES sourceFileAttr;

      InitializeObjectAttributes(&sourceFileAttr, FileName,

                  OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr,
nullptr);

      status = FltCreateFile(

                  FltObjects->Filter,

                  FltObjects->Instance,

                  &hSourceFile,

                  FILE_READ_DATA | SYNCHRONIZE,

                  &sourceFileAttr,

                  &ioStatus,

                  nullptr, FILE_ATTRIBUTE_NORMAL,

                  FILE_SHARE_READ | FILE_SHARE_WRITE,

                  FILE_OPEN,

                  FILE_SYNCHRONOUS_IO_NONALERT,
```

```
                nullptr, 0,
                IO_IGNORE_SHARE_ACCESS_CHECK);
    if (!NT_SUCCESS(status))
        break;
```

Перед вызовом FltCreateFile, как и другим API, требующим объект OBJECT_ATTRIBUTES, структура должна быть правильно инициализирована с именем файла, предоставленным в BackupFile.

Это файловый поток по умолчанию, который вот-вот изменится в результате операции записи, и поэтому мы делаем резервную копию.

Важными аргументами вызова являются:

• объекты filter и instance, которые предоставляют необходимую информацию для перехода вызова к следующему фильтру нижнего уровня (или файловой системы), вместо перехода на вершину стека файловой системы.

• возвращаемый дескриптор в hSourceFile.

• маска доступа, установленная на FILE_READ_DATA и SYNCHRONIZE.

• расположение создания, в данном случае указывающее, что файл должен существовать (FILE_OPEN).

• для параметров создания установлено значение FILE_SYNCHRONOUS_IO_NONALERT, что указывает на синхронную операцию.

• флаг IO_IGNORE_SHARE_ACCESS_CHECK важен, потому что рассматриваемый файл был уже открыт клиентом, который, скорее всего, открыл его без разрешения на совместное использование.

Затем нам нужно открыть или создать поток резервных копий в том же файле. Назовем резервную копию “:backup“ и используйте другой вызов FltCreateFile, чтобы получить дескриптор целевого файла:

```
UNICODE_STRING targetFileName;
const WCHAR backupStream[] = L":backup";
targetFileName.MaximumLength = FileName->Length + sizeof(backupStream);
targetFileName.Buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool,
            targetFileName.MaximumLength, DRIVER_TAG);
if (targetFileName.Buffer == nullptr)
      return STATUS_INSUFFICIENT_RESOURCES;
RtlCopyUnicodeString(&targetFileName, FileName);
RtlAppendUnicodeToString(&targetFileName, backupStream);
OBJECT_ATTRIBUTES targetFileAttr;
InitializeObjectAttributes(&targetFileAttr, &targetFileName,
            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);
status = FltCreateFile(
            FltObjects->Filter,
            FltObjects->Instance,
            &hTargetFile,
            GENERIC_WRITE | SYNCHRONIZE, // access mask
            &targetFileAttr,
            &ioStatus,
            nullptr, FILE_ATTRIBUTE_NORMAL,
            0,
            FILE_OVERWRITE_IF,
            FILE_SYNCHRONOUS_IO_NONALERT, // create options (sync I/O)
            nullptr, 0, 0);
// extended attributes, EA length, flags
ExFreePool(targetFileName.Buffer);
if (!NT_SUCCESS(status))
      break;
```

Имя файла создается путем объединения основного имени файла и имени резервного потока. Он открыт для доступа на запись (GENERIC_WRITE) и перезаписывает любые данные, которые могут присутствовать (FILE_OVERWRITE_IF).

С помощью этих двух дескрипторов мы можем читать с источника и писать в цель. Простой подход было бы выделить буфер с размером файла и выполнить работу с одним чтением и одной записью.

Однако это может быть проблематично, если файл очень большой, что может вызвать нехватки памяти.

Решением было бы выделить относительно небольшой буфер и просто зацикливаться, пока все куски файла не были скопированы. Это подход, который мы будем использовать. Сначала выделим буфер:

```
ULONG size = 1 << 21; // 2 MB

buffer = ExAllocatePoolWithTag(PagedPool, size, DRIVER_TAG);

if (!buffer) {

status = STATUS_INSUFFICIENT_RESOURCES;

break;

}
```

Теперь цикл:

```
LARGE_INTEGER offset = { 0 };

LARGE_INTEGER writeOffset = { 0 };

ULONG bytes;

auto saveSize = fileSize;

while (fileSize.QuadPart > 0) {

      status = ZwReadFile(

                  hSourceFile,

                  nullptr,

                  nullptr, nullptr,

                  &ioStatus,

                  buffer,

                  (ULONG)min((LONGLONG)size, fileSize.QuadPart),

                  &offset,

                  nullptr);

      if (!NT_SUCCESS(status))
```

```
            break;
        bytes = (ULONG)ioStatus.Information;
        status = ZwWriteFile(
                    hTargetFile,
                    nullptr,
                    nullptr, nullptr, // APC routine, APC context
                    &ioStatus,
                    buffer,
                    bytes,
                    &writeOffset,
                    nullptr);
        if (!NT_SUCCESS(status))
            break;
        offset.QuadPart += bytes;
        writeOffset.QuadPart += bytes;
        fileSize.QuadPart -= bytes;
}
```

Цикл продолжается до тех пор, пока есть байты для передачи. Начнем с размера файла, а затем уменьшаем его для каждого переданного фрагмента.

Саму работу выполняют функции ZwReadFile и ZwWriteFile.

Операция чтения возвращает количество фактических байтов, эти байты мы потом должны записать.

Когда все будет сделано, осталось сделать еще одно. Поскольку мы можем перезаписывать предыдущую резервную копию (которая могла быть больше, чем текущая), мы должны установить указатель конца файла на текущее смещение:

```
FILE_END_OF_FILE_INFORMATION info;
info.EndOfFile = saveSize;
NT_VERIFY(NT_SUCCESS(ZwSetInformationFile(hTargetFile, &ioStatus,
&info, sizeof(info), FileEndOfFileInformation)));
} while (false);
```

Макрос NT_VERIFY работает как NT_ASSERT в отладочных сборках, но не отбрасывает в релизных сборках.

Наконец, нам нужно все очистить:

```
if (buffer)

ExFreePool(buffer);

if (hSourceFile)

FltClose(hSourceFile);

if (hTargetFile)

FltClose(hTargetFile);

return status;

}
```

Обратный вызов после очистки

Зачем нужен еще один обратный вызов? Наш контекст прикреплен к файлу, что означает, что он будет удаляться только при удалении файла, что может никогда не произойти.

Нам нужно освободить контекст, когда файл закрывается клиентом.

Здесь важны две операции: IRP_MJ_CLOSE и IRP_MJ_CLEANUP.

Лучший подход — использовать IRP_MJ_CLEANUP, что по сути означает, что файловый объект больше не нужен, даже если последний дескриптор еще не закрыт. Это хорошее время, чтобы освободить наш контекст (если он существует).

```
FLT_POSTOP_CALLBACK_STATUS FileBackupPostCleanup(

            PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects,

            PVOID CompletionContext, FLT_POST_OPERATION_FLAGS Flags) {

      UNREFERENCED_PARAMETER(Flags);

      UNREFERENCED_PARAMETER(CompletionContext);

      UNREFERENCED_PARAMETER(Data);
```

Нам нужно получить контекст файла, и если он существует - освободить все, что динамически выделено непосредственно перед удалением:

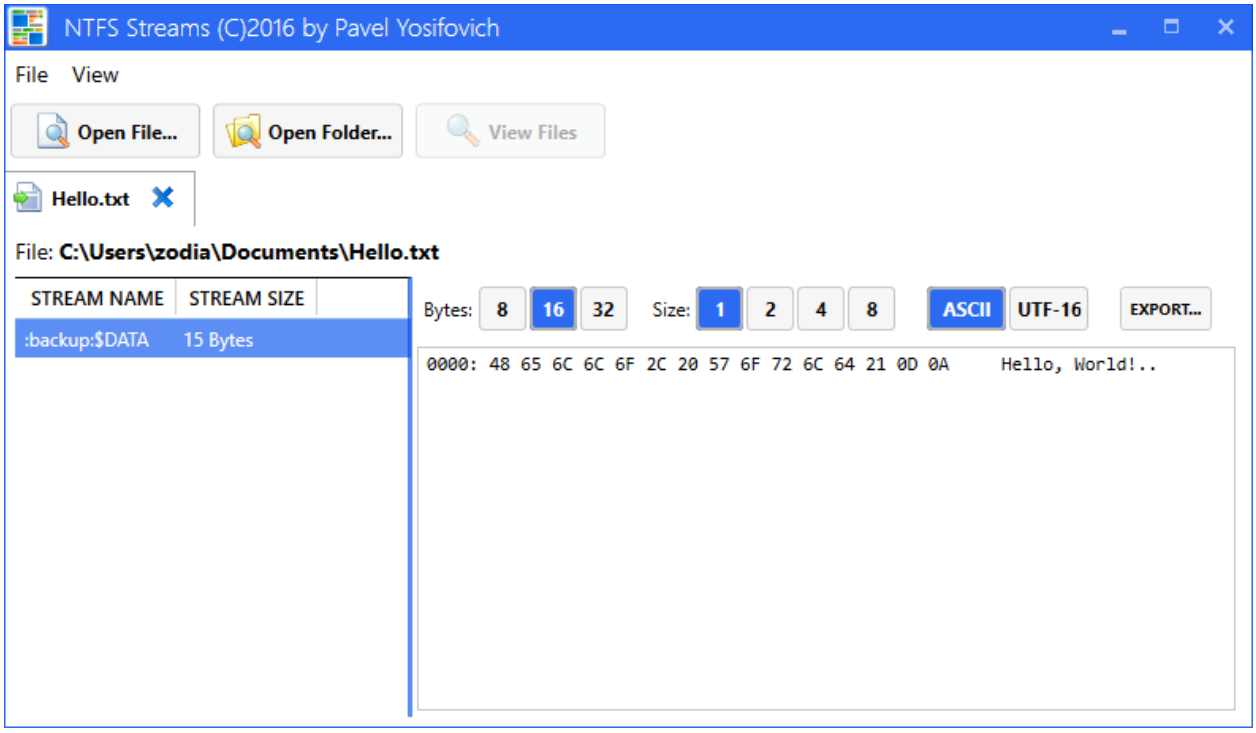
```
FileContext* context;
auto status = FltGetFileContext(FltObjects->Instance,
            FltObjects->FileObject, (PFLT_CONTEXT*)&context);
if (!NT_SUCCESS(status) || context == nullptr) {
      // no context, continue normally
      return FLT_POSTOP_FINISHED_PROCESSING;
}
if (context->FileName.Buffer)
      ExFreePool(context->FileName.Buffer);
FltReleaseContext(context);
FltDeleteContext(context);
return FLT_POSTOP_FINISHED_PROCESSING;
}
```

Тестирование драйвера

Мы можем протестировать драйвер, развернув его в целевой системе как обычно, а затем работать с файлами в каталогах «documents» или «pictures».

В следующем примере я создал файл hello.txt в папке документов с содержимым «Hello, world!», сохранил файл, а затем изменил его содержимое на« Goodbye, world! » и снова сохранил.

Рисунок ниже показывает, что произошло в NtfsStreams:



Восстановление резервных копий

Как мы можем восстановить резервную копию? Нам нужно скопировать содержимое потока «: backup» поверх «обычного» файла.

К сожалению, API CopyFile не может этого сделать, так как не принимает альтернативные потоки.

Давайте напишем утилиту для работы.

Мы создадим новый проект консольного приложения с именем FileRestore. Мы добавим следующие заголовочные файлы в pch.h:

```
#include <Windows.h>
#include <stdio.h>
#include <string>
```

Основная функция должна принимать имя файла в качестве аргумента командной строки:

```
int wmain(int argc, const wchar_t* argv[]) {
	if (argc < 2) {
		printf("Usage: FileRestore <filename>\n");
		return 0;
	}
```

Затем мы откроем два файла, один из которых указывает на поток «: backup», а другой - на «обычный» файл.

Затем мы скопируем по частям, как и код BackupFile драйвера, но в пользовательском режиме (Код обработки ошибок опущен для краткости):

```
// generate full stream name
std::wstring stream(argv[1]);
stream += L":backup";
HANDLE hSource = ::CreateFile(stream.c_str(), GENERIC_READ, FILE_SHARE_READ,
		nullptr, OPEN_EXISTING, 0, nullptr);
HANDLE hTarget = ::CreateFile(argv[1], GENERIC_WRITE, 0,
		nullptr, OPEN_EXISTING, 0, nullptr);
LARGE_INTEGER size;
::GetFileSizeEx(hSource, &size);
ULONG bufferSize = (ULONG)min((LONGLONG)1 << 21, size.QuadPart);
void* buffer = VirtualAlloc(nullptr, bufferSize,
```

```
            MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
DWORD bytes;
while (size.QuadPart > 0) {
      ::ReadFile(hSource, buffer,
                  (DWORD)(min((LONGLONG)bufferSize, size.QuadPart)),
                  &bytes, nullptr);
      ::WriteFile(hTarget, buffer, bytes, &bytes, nullptr);
      size.QuadPart -= bytes;
}
printf("Restore successful!\n");
::CloseHandle(hSource);
::CloseHandle(hTarget);
::VirtualFree(buffer, 0, MEM_DECOMMIT | MEM_RELEASE);
return 0;
}
```

Связь в пользовательском режиме

В предыдущих главах мы видели один из способов связи между драйвером и клиентом пользовательского режима с помощью DeviceIoControl. Это, безусловно, прекрасный способ, который хорошо работает во многих ситуациях.

Один из его недостатков является то, что клиент пользовательского режима должен инициировать обмен данными. Если в драйвере что-то есть для отправки клиенту (или клиентам) пользовательского режима, он не может делать это напрямую.

Он должен хранить его и ждать пока клиент запросит данные.

Диспетчер фильтров предоставляет альтернативный механизм для двунаправленной связи между мини-фильтром файловой системы и клиентом пользовательского режима, где любая сторона может отправлять информацию другой и даже ждать ответа.

Мини-фильтр создает порт связи фильтра, вызывая FltCreateCommunicationPort.

Клиент пользовательского режима подключается к порту, вызывая FilterConnectCommunicationPort, получая дескриптор порта.

Мини-фильтр отправляет сообщение своему клиенту(-ам) пользовательского режима с FltSendMessage.

И наоборот, клиент пользовательского режима вызывает FilterGetMessage, чтобы дождаться прибытия сообщения, или вызывает FilterSendMessage что-бы отправить

сообщение драйверу. Если драйвер ожидает ответа, клиент пользовательского режима вызывает FilterReplyMessage с ответом.

Создание коммуникационного порта

Функция FltCreateCommunicationPort объявляется следующим образом:

```
NTSTATUS FltCreateCommunicationPort (
            _In_ PFLT_FILTER Filter,
            _Outptr_ PFLT_PORT *ServerPort,
            _In_ POBJECT_ATTRIBUTES ObjectAttributes,
            _In_opt_ PVOID ServerPortCookie,
            _In_ PFLT_CONNECT_NOTIFY ConnectNotifyCallback,
            _In_ PFLT_DISCONNECT_NOTIFY DisconnectNotifyCallback,
            _In_opt_ PFLT_MESSAGE_NOTIFY MessageNotifyCallback,
            _In_ LONG MaxConnections);
```

Вот описание параметров FltCreateCommunicationPort:

• `Filter` - это указатель, возвращаемый FltRegisterFilter.

• ServerPort - это выходной дескриптор, который используется внутри для прослушивания входящих сообщений из пользовательского режима.

• ObjectAttributes - это стандартная структура атрибутов, которая должна содержать имя порта сервера и дескриптор безопасности, который позволит подключаться клиентам пользовательского режима (подробнее об этом позже).

• ServerPortCookie - это необязательный указатель, определяемый драйвером, который можно использовать для различения несколько открытых портов в обратных вызовах сообщений.

• ConnectNotifyCallback - это обратный вызов, который должен предоставить драйвер, который вызывается при подключении нового клиента.

• DisconnectNotifyCallback - это обратный вызов, вызываемый, когда клиент пользовательского режима отключается от порта.

• MessageNotifyCallback - это обратный вызов, вызываемый при поступлении сообщения на порт.

• MaxConnections указывает максимальное количество клиентов, которые могут подключиться к порту. Это должно быть больше нуля.

Для успешного вызова FltCreateCommunicationPort драйвер должен подготовить атрибуты объекта и дескриптор безопасности. Самый простой дескриптор безопасности можно создать с помощью FltBuildDefaultSecurityDescriptor так:

```
PSECURITY_DESCRIPTOR sd;

status = FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
```

Затем можно инициализировать атрибуты объекта:

```
UNICODE_STRING portName = RTL_CONSTANT_STRING(L"\\MyPort");

OBJECT_ATTRIBUTES portAttr;

InitializeObjectAttributes(&portAttr, &name,

OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, sd);
```

Имя порта находится в пространстве имен диспетчера объектов, которое можно просмотреть с помощью WinObj после создания порта.

Флаги должны включать OBJ_KERNEL_HANDLE, иначе вызов завершится неудачно. Обратите внимание, что последний аргумент дескриптор безопасности, определенный ранее.

Теперь драйвер готов вызвать FltCreateCommunicationPort, как правило, выполняется после того, как драйвер вызывает FltRegisterFilter, но перед FltStartFiltering:

```
PFLT_PORT ServerPort;

status = FltCreateCommunicationPort(FilterHandle, &ServerPort, &portAttr,
nullptr,

PortConnectNotify, PortDisconnectNotify, PortMessageNotify, 1);

// free security descriptor

FltFreeSecurityDescriptor(sd);
```

Подключение в пользовательском режиме

Клиенты пользовательского режима вызывают FilterConnectCommunicationPort для подключения к открытому порту, объявленному вот так:

```
HRESULT FilterConnectCommunicationPort (

            _In_ LPCWSTR lpPortName,

            _In_ DWORD dwOptions,

            _In_reads_bytes_opt_(wSizeOfContext) LPCVOID lpContext,

            _In_ WORD wSizeOfContext,

            _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,

            _Outptr_ HANDLE *hPort);
```

Вот краткое изложение параметров:

• lpPortName - это имя порта (например, «\MyPort»). Обратите внимание, что с дескриптором безопасности по умолчанию, могут подключаться только процессы уровня администратора.

• dwOptions обычно равен нулю, но FLT_PORT_FLAG_SYNC_HANDLE в Windows 8.1 и новее указывает, что возвращаемый дескриптор должен работать только синхронно. Непонятно, почему это необходимо, так как использование по умолчанию в любом случае синхронно.

• lpContext и wSizeOfContext поддерживают способ отправки буфера драйверу во время соединения. Это может быть использовано как средство аутентификации, например, когда какой-либо пароль или токен отправляется драйверу, и драйвер не принимает запросы на подключение, которые не соответствуют некоторым предопределенный механизм аутентификации.

В производственном драйвере это обычно хорошая идея, поэтому что неизвестные клиенты не могли «захватить» коммуникационный порт у законных клиентов.

• lpSecurityAttributes - это обычный пользовательский режим SECURITY_ATTRIBUTES, обычно установленный в NULL.

• hPort - дескриптор вывода, который позже используется клиентом для отправки и получения сообщений.

Этот вызов вызывает обратный вызов уведомления о подключении клиента, объявленный следующим образом:

```
NTSTATUS PortConnectNotify(
            _In_ PFLT_PORT ClientPort,
            _In_opt_ PVOID ServerPortCookie,
            _In_reads_bytes_opt_(SizeOfContext) PVOID ConnectionContext,
            _In_ ULONG SizeOfContext,
            _Outptr_result_maybenull_ PVOID *ConnectionPortCookie);
```

ClientPort - это уникальный дескриптор порта клиента, который драйвер должен хранить и использовать всякий раз, когда ему нужно общаться с этим клиентом.

ServerPortCookie - это тот же параметр, клоторый указан в FltCreateCommunicationPort.

Параметры ConnectionContex и SizeOfContex содержат необязательный буфер, отправленный клиентом.

Наконец, ConnectionPortCookie - необязательное значение, он передается при отключении клиента и сообщении процедуры уведомления.

Если драйвер соглашается принять соединение клиента, он возвращает STATUS_SUCCESS. В противном случае клиент получит сообщение об ошибке.

После успешного вызова FilterConnectCommunicationPort клиент может начать общение с драйвером и наоборот.

Отправка и получение сообщений

Драйвер мини-фильтра может отправлять сообщения клиентам с FltSendMessage, объявленным следующим образом:

```
NTSTATUS

FLTAPI

FltSendMessage (

            _In_ PFLT_FILTER Filter,

            _In_ PFLT_PORT *ClientPort,

            _In_ PVOID SenderBuffer,

            _In_ ULONG SenderBufferLength,

            _Out_ PVOID ReplyBuffer,

            _Inout_opt_ PULONG ReplyLength,

            _In_opt_ PLARGE_INTEGER Timeout);
```

К настоящему времени должны быть известны первые два параметра. Драйвер может отправлять любой буфер, описанный SenderBuffer с длиной SenderBufferLength.

Обычно драйвер определяет некоторую структуру в общий файл заголовка, который клиент также может включить, чтобы правильно интерпретировать полученный буфер. При желании драйвер может ожидать ответа, и если да, параметр ReplyBuffer должен быть не- NULL с максимальной длиной ответа, хранящейся в ReplyLength.

Наконец, тайм-аут указывает, как долго драйвер готов ждать сообщения, чтобы добраться до клиента . Тайм-аут имеет обычный формат, описанный здесь для удобства:

• если указатель NULL, драйвер готов ждать бесконечно.

• если значение положительное, то это абсолютное время в единицах 100 нс с 1 января 1601 г.

• если значение отрицательное, это относительное время - наиболее распространенный случай - в тех же единицах 100 нс.

Например, чтобы указать одну секунду, укажите -100000000. В качестве другого примера, чтобы указать в миллисекундах умножьте x на -10000.

Драйвер должен быть осторожен и не указывать NULL из обратного вызова, потому что это означает, что если клиент в настоящее время не слушает, поток блокируется до тех пор, пока это не произойдет, что может никогда не произойти.

Лучше указать некоторое ограниченного значение.

С точки зрения клиента, он может ждать сообщения от драйвера с помощью FilterGetMessage, который использует структуру FILTER_MESSAGE_HEADER:

```
typedef struct _FILTER_MESSAGE_HEADER {

ULONG ReplyLength;

ULONGLONG MessageId;

} FILTER_MESSAGE_HEADER, *PFILTER_MESSAGE_HEADER;
```

Если ожидается ответ, ReplyLength указывает максимальное ожидаемое количество байтов.

MessageId поле позволяет различать сообщения, которые клиент должен использовать, если он вызывает FilterReplyMessage.

Клиент может инициировать собственное сообщение с помощью FilterSendMessage, которое в конечном итоге попадает в обратный вызов драйвера, который зарегистрирован в FltCreateCommunicationPort.

FilterSendMessage может указывать буфер, содержащий сообщение для отправки и необязательный буфер для ожидаемого ответа из мини-фильтра.

Расширенный драйвер резервного копирования файлов

Давайте усовершенствуем драйвер резервного копирования файлов, чтобы отправлять уведомления клиенту пользовательского режима, когда резервная копия была сделана.

Сначала мы определим некоторую глобальную переменную для хранения состояния, связанного с портом связи:

PFLT_PORT FilterPort;

PFLT_PORT SendClientPort;

FilterPort - это порт сервера драйвера, а SendClientPort - это клиентский порт после подключения (мы будем разрешать только одного клиента).

Нам придется изменить DriverEntry, чтобы создать коммуникационный порт, как описано в предыдущем разделе.Вот код после успешного выполнения FltRegisterFilter без обработки ошибок:

```
UNICODE_STRING name = RTL_CONSTANT_STRING(L"\\FileBackupPort");

PSECURITY_DESCRIPTOR sd;

status = FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);

OBJECT_ATTRIBUTES attr;

InitializeObjectAttributes(&attr, &name,

            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, sd);

status = FltCreateCommunicationPort(gFilterHandle, &FilterPort, &attr,

            nullptr, PortConnectNotify, PortDisconnectNotify,
PortMessageNotify, 1);

FltFreeSecurityDescriptor(sd);

//Start filtering i/o

status = FltStartFiltering(gFilterHandle);
```

Драйвер позволяет подключаться к порту только одному клиенту, что довольно часто встречается, когда мини-фильтр работает в тандеме с сервисом пользовательского режима.

Обратный вызов PortConnectNotify вызывается, когда клиент пытается подключиться. Наш драйвер просто сохраняет порт клиента и возвращает успех:

```
_Use_decl_annotations_

NTSTATUS PortConnectNotify(

            PFLT_PORT ClientPort, PVOID ServerPortCookie, PVOID
ConnectionContext,

            ULONG SizeOfContext, PVOID* ConnectionPortCookie) {

      UNREFERENCED_PARAMETER(ServerPortCookie);

      UNREFERENCED_PARAMETER(ConnectionContext);

      UNREFERENCED_PARAMETER(SizeOfContext);

      UNREFERENCED_PARAMETER(ConnectionPortCookie);

      SendClientPort = ClientPort;

      return STATUS_SUCCESS;

}
```

Когда клиент отключается, вызывается обратный вызов PortDisconnectNotify. Важно закрыть порт клиента в это время, иначе мини-фильтр никогда не будет выгружен:

```
void PortDisconnectNotify(PVOID ConnectionCookie) {
	UNREFERENCED_PARAMETER(ConnectionCookie);
	FltCloseClientPort(gFilterHandle, &SendClientPort);
	SendClientPort = nullptr;
}
```

В этом драйвере мы не ожидаем никаких сообщений от клиента - только драйвер отправляет сообщения - поэтому обратный вызов PostMessageNotify имеет пустую реализацию.

Теперь нам нужно отправить сообщение об успешном резервном копировании файла.

Для этой цели мы определим структуру сообщения, общую для драйвера и клиента, в собственном файле заголовка FileBackupCommon.h:

```
struct FileBackupPortMessage {
USHORT FileNameLength;
WCHAR FileName[1];
};
```

Сообщение содержит длину имени файла и само имя файла. Сообщение не имеет фиксированный размер и зависит от длины имени файла. В обратном вызове перед записью после резервного копирования файла нам нужно выделить и инициализировать буфер для отправки:

```
if (SendClientPort) {
USHORT nameLen = context->FileName.Length;
USHORT len = sizeof(FileBackupPortMessage) + nameLen;
auto msg = (FileBackupPortMessage*)ExAllocatePoolWithTag(PagedPool, len,
DRIVER_TAG);
if (msg) {
msg->FileNameLength = nameLen / sizeof(WCHAR);
RtlCopyMemory(msg->FileName, context->FileName.Buffer, nameLen);
```

Сначала мы проверяем, подключен ли какой-либо клиент, и если да, мы выделяем буфер нужного размера для включения имя файла.

Затем копируем его в буфер (RtlCopyMemory, то же, что и memcpy).

Теперь мы готовы отправить сообщение с ограниченным временем ожидания:

```
LARGE_INTEGER timeout;

timeout.QuadPart = -10000 * 100; // 100msec

FltSendMessage(gFilterHandle, &SendClientPort, msg, len,

nullptr, nullptr, &timeout);

ExFreePool(msg);

}
```

Наконец, в процедуре выгрузки фильтра мы должны закрыть порт связи фильтра (перед FltUnregisterFilter):

FltCloseCommunicationPort(FilterPort);

Клиент пользовательского режима

Давайте создадим простой клиент, который открывает порт и прослушивает сообщения файлов, для которых выполняется резервное копирование.

Создайте новое консольное приложение с именем FileBackupMon. В pch.h мы добавляем следующие #include:

```
#include <Windows.h>

#include <fltUser.h>

#include <stdio.h>

#include <string>
```

fltuser.h - заголовок пользовательского режима, в котором объявлены функции FilterXxx (они не являются частью из windows.h). В файле cpp мы должны добавить библиотеку импорта, в которой находятся эти функции.

Вот добавление этой библиотеке:

#pragma comment(lib, "fltlib")

Наша основная функция должна сначала открыть коммуникационный порт:

```
HANDLE hPort;

auto hr = ::FilterConnectCommunicationPort(L"\\FileBackupPort",

            0, nullptr, 0, nullptr, &hPort);

if (FAILED(hr)) {

      printf("Error connecting to port (HR=0x%08X)\n", hr);

      return 1;

}
```

Теперь мы можем выделить буфер для входящих сообщений и бесконечно зацикливаться на ожидании сообщений.

Как только сообщение будет получено, мы отправим его на обработку:

```
BYTE buffer[1 << 12]; // 4 KB

auto message = (FILTER_MESSAGE_HEADER*)buffer;

for (;;) {

hr = ::FilterGetMessage(hPort, message, sizeof(buffer), nullptr);

if (FAILED(hr)) {

printf("Error receiving message (0x%08X)\n", hr);

break;

}

HandleMessage(buffer + sizeof(FILTER_MESSAGE_HEADER));

}
```

Буфер здесь выделяется статически, потому что сообщение просто включает в основном имя файла, поэтому буфера 4KB более чем достаточно. Как только сообщение получено, мы передаем тело сообщения в функцию HandleMessage, стараясь пропустить всегда присутствующий заголовок.

Теперь осталось что-то сделать с данными:

```
void HandleMessage(const BYTE* buffer) {

      auto msg = (FileBackupPortMessage*)buffer;

      std::wstring filename(msg->FileName, msg->FileNameLength);

      printf("file backed up: %ws\n", filename.c_str());

}
```

Отладка

Отладка Мини-фильтра файловой системы ничем не отличается от отладки любого другого драйвера ядра.

Тем не менее, пакет Debugging Tools for Windows имеет специальное расширение DLL, fltkd.dll, со специфическим командами. Эта DLL не является одной из загружаемых по умолчанию расширений DLL, поэтому команды должны использоваться с их «полным именем», которое включает префикс fltkd и команду.

В качестве альтернативы, DLL можно загрузить явно с помощью команды .load, а затем команды можно использовать напрямую.

В следующей таблице показаны некоторые команды fltkd с кратким описанием.

Command	Description
!help	shows the command list with brief descriptions
!filters	shows information on all loaded mini-filters
!filter	shows information for the specified filter address
!instance	shows information for the specified instance address
!volumes	shows all volume objects
!volume	shows detailed information on the specified volume address
!portlist	shows the server ports for the specified filter
!port	shows information on the specified client port

Вот пример сеанса с использованием некоторых из вышеперечисленных команд:

```
2: kd> .load fltkd
2: kd> !filters

Filter List: ffff8b8f55bf60c0 "Frame 0"
   FLT_FILTER: ffff8b8f579d9010 "bindflt" "409800"
      FLT_INSTANCE: ffff8b8f62ea8010 "bindflt Instance" "409800"
   FLT_FILTER: ffff8b8f5ba06010 "CldFlt" "409500"
      FLT_INSTANCE: ffff8b8f550aaa20 "CldFlt" "180451"
   FLT_FILTER: ffff8b8f55ceca20 "WdFilter" "328010"
      FLT_INSTANCE: ffff8b8f572d6b30 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f575d5b30 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f585d2050 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f58bde010 "WdFilter Instance" "328010"
   FLT_FILTER: ffff8b8f5cdc6320 "storqosflt" "244000"
   FLT_FILTER: ffff8b8f550aca20 "wcifs" "189900"
      FLT_INSTANCE: ffff8b8f551a6720 "wcifs Instance" "189900"
   FLT_FILTER: ffff8b8f576cab30 "FileCrypt" "141100"
   FLT_FILTER: ffff8b8f550b2010 "luafv" "135000"
      FLT_INSTANCE: ffff8b8f550ae010 "luafv" "135000"
```

```
   FLT_FILTER: ffff8b8f633e8c80 "FileBackup" "100200"
      FLT_INSTANCE: ffff8b8f645df290 "FileBackup Instance" "100200"
      FLT_INSTANCE: ffff8b8f5d1a7880 "FileBackup Instance" "100200"
   FLT_FILTER: ffff8b8f58ce2be0 "npsvctrig" "46000"
      FLT_INSTANCE: ffff8b8f55113a60 "npsvctrig" "46000"
   FLT_FILTER: ffff8b8f55ce9010 "Wof" "40700"
      FLT_INSTANCE: ffff8b8f572e2b30 "Wof Instance" "40700"
      FLT_INSTANCE: ffff8b8f5bae7010 "Wof Instance" "40700"
   FLT_FILTER: ffff8b8f55ce8520 "FileInfo" "40500"
      FLT_INSTANCE: ffff8b8f579cea20 "FileInfo" "40500"
      FLT_INSTANCE: ffff8b8f577ee8a0 "FileInfo" "40500"
      FLT_INSTANCE: ffff8b8f58cc6730 "FileInfo" "40500"
      FLT_INSTANCE: ffff8b8f5bae2010 "FileInfo" "40500"
2: kd> !portlist ffff8b8f633e8c80

FLT_FILTER: ffff8b8f633e8c80
   Client Port List         : Mutex (ffff8b8f633e8ed8) List [ffff8b8f5949b7a0-ffff8b\
8f5949b7a0] mCount=1
      FLT_PORT_OBJECT: ffff8b8f5949b7a0
         FilterLink              : [ffff8b8f633e8f10-ffff8b8f633e8f10]
         ServerPort              : ffff8b8f5b195200
         Cookie                  : 0000000000000000
         Lock                    : (ffff8b8f5949b7c8)
         MsgQ                    : (ffff8b8f5949b800)  NumEntries=1 Enabled
         MessageId               : 0x0000000000000000
         DisconnectEvent         : (ffff8b8f5949b8d8)
         Disconnected            : FALSE

2: kd> !volumes

Volume List: ffff8b8f55bf6140 "Frame 0"
   FLT_VOLUME: ffff8b8f579cb6b0 "\Device\Mup"
      FLT_INSTANCE: ffff8b8f572d6b30 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f579cea20 "FileInfo" "40500"
   FLT_VOLUME: ffff8b8f57af8530 "\Device\HarddiskVolume4"
      FLT_INSTANCE: ffff8b8f62ea8010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffff8b8f575d5b30 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f551a6720 "wcifs Instance" "189900"
      FLT_INSTANCE: ffff8b8f550aaa20 "CldFlt" "180451"
      FLT_INSTANCE: ffff8b8f550ae010 "luafv" "135000"
      FLT_INSTANCE: ffff8b8f645df290 "FileBackup Instance" "100200"
      FLT_INSTANCE: ffff8b8f572e2b30 "Wof Instance" "40700"
      FLT_INSTANCE: ffff8b8f577ee8a0 "FileInfo" "40500"
```

```
   FLT_VOLUME: ffff8b8f58cc4010 "\Device\NamedPipe"
      FLT_INSTANCE: ffff8b8f55113a60 "npsvctrig" "46000"
   FLT_VOLUME: ffff8b8f58ce8060 "\Device\Mailslot"
   FLT_VOLUME: ffff8b8f58ce1370 "\Device\HarddiskVolume2"
      FLT_INSTANCE: ffff8b8f585d2050 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f58cc6730 "FileInfo" "40500"
   FLT_VOLUME: ffff8b8f5b227010 "\Device\HarddiskVolume1"
      FLT_INSTANCE: ffff8b8f58bde010 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f5d1a7880 "FileBackup Instance" "100200"
      FLT_INSTANCE: ffff8b8f5bae7010 "Wof Instance" "40700"
      FLT_INSTANCE: ffff8b8f5bae2010 "FileInfo" "40500"

2: kd> !volume ffff8b8f57af8530

FLT_VOLUME: ffff8b8f57af8530 "\Device\HarddiskVolume4"
   FLT_OBJECT: ffff8b8f57af8530  [04000000] Volume
      RundownRef               : 0x00000000000008b2 (1113)
      PointerCount             : 0x00000001
      PrimaryLink              : [ffff8b8f58cc4020-ffff8b8f579cb6c0]
   Frame                    : ffff8b8f55bf6010 "Frame 0"
   Flags                    : [00000164] SetupNotifyCalled EnableNameCaching FilterA\
ttached +100!!
   FileSystemType           : [00000002] FLT_FSTYPE_NTFS
   VolumeLink               : [ffff8b8f58cc4020-ffff8b8f579cb6c0]
   DeviceObject             : ffff8b8f573cab60
   DiskDeviceObject         : ffff8b8f572e7b80
   FrameZeroVolume          : ffff8b8f57af8530
   VolumeInNextFrame        : 0000000000000000
   Guid                     : "\??\Volume{5379a5de-f305-4243-a3ec-311938a2df19}"
   CDODeviceName            : "\Ntfs"
   CDODriverName            : "\FileSystem\Ntfs"
   TargetedOpenCount        : 1104
   Callbacks                : (ffff8b8f57af8650)
   ContextLock              : (ffff8b8f57af8a38)
   VolumeContexts           : (ffff8b8f57af8a40)  Count=0
   StreamListCtrls          : (ffff8b8f57af8a48)  rCount=29613
   FileListCtrls            : (ffff8b8f57af8ac8)  rCount=22668
   NameCacheCtrl            : (ffff8b8f57af8b48)
   InstanceList             : (ffff8b8f57af85d0)
      FLT_INSTANCE: ffff8b8f62ea8010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffff8b8f575d5b30 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff8b8f551a6720 "wcifs Instance" "189900"
      FLT_INSTANCE: ffff8b8f550aaa20 "CldFlt" "180451"
```

```
      FLT_INSTANCE: ffff8b8f550ae010 "luafv" "135000"
      FLT_INSTANCE: ffff8b8f645df290 "FileBackup Instance" "100200"
      FLT_INSTANCE: ffff8b8f572e2b30 "Wof Instance" "40700"
      FLT_INSTANCE: ffff8b8f577ee8a0 "FileInfo" "40500"

2: kd> !instance ffff8b8f5d1a7880

FLT_INSTANCE: ffff8b8f5d1a7880 "FileBackup Instance" "100200"
   FLT_OBJECT: ffff8b8f5d1a7880  [01000000] Instance
      RundownRef              : 0x0000000000000000 (0)
      PointerCount            : 0x00000001
      PrimaryLink             : [ffff8b8f5bae7020-ffff8b8f58bde020]
   OperationRundownRef      : ffff8b8f639c61b0
      Number                  : 3
      PoolToFree              : ffff8b8f65aad590
      OperationsRefs          : ffff8b8f65aad5c0  (0)
         PerProcessor Ref[0]      : 0x0000000000000000 (0)
         PerProcessor Ref[1]      : 0x0000000000000000 (0)
         PerProcessor Ref[2]      : 0x0000000000000000 (0)
   Flags                   : [00000000]
   Volume                  : ffff8b8f5b227010 "\Device\HarddiskVolume1"
   Filter                  : ffff8b8f633e8c80 "FileBackup"
   TrackCompletionNodes    : ffff8b8f5f3f3cc0
   ContextLock             : (ffff8b8f5d1a7900)
   Context                 : 0000000000000000
   CallbackNodes           : (ffff8b8f5d1a7920)
   VolumeLink              : [ffff8b8f5bae7020-ffff8b8f58bde020]
   FilterLink              : [ffff8b8f633e8d50-ffff8b8f645df300]
```

Упражнения

1. Напишите мини-фильтр файловой системы, который перехватывает операции удаления из cmd.exe и удаляя их, файлы перемещаются в корзину.

2. Расширьте драйвер резервного копирования файлов возможностью выбора каталогов, в которых будут храниться резервные копии.

3. Расширьте драйвер резервного копирования файлов, включив в него несколько резервных копий, ограниченных каким-либо правилом, например, размером файла, датой или максимальное количество резервных копий.

4. Измените драйвер резервного копирования файлов для резервного копирования только измененных данных, а не всего файла.

5. Придумайте собственные идеи для драйвера мини-фильтра файловой системы!

Резюме

Эта глава была посвящена мини-фильтрам файловой системы - мощным драйверам, способным перехватывать любую деятельность файловой системы.

Мини-фильтры - большая тема, и эта глава должна помочь вам начать интересное путешествие в мир виртуальной файловой системы.

Вы можете найти дополнительную информацию в документации WDK.

Примеры WDK на Github и в некоторых блогах.

В следующей (и последней) главе мы рассмотрим различные методы разработки драйверов и другие разные темы, которые пока не подходили к главам.