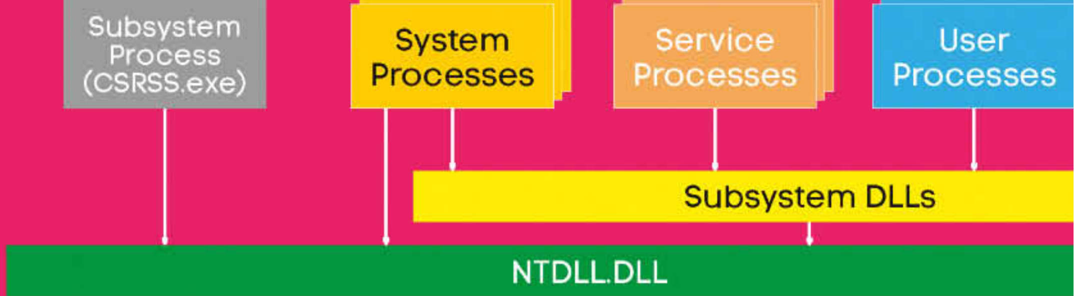


РАБОТА С ЯДРОМ Windows

ПАВЕЛ ЙОСИФОВИЧ

Executive

Win32k.Sys

Device Drivers

Kernel

Hardware Abstraction Layer (HAL)

Windows Kernel Programming

Pavel Yosifovich

ПАВЕЛ ЙОСИФОВИЧ

РАБОТА С ЯДРОМ Windows

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2021

ББК 32.973.2-018.2
УДК 004.451
И75

Йосифович Павел

И75 Работа с ядром Windows. — СПб.: Питер, 2021. — 400 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1680-5

Ядро Windows таит в себе большую силу. Но как заставить ее работать? Павел Йосифович поможет вам справиться с этой сложной задачей: пояснения и примеры кода превратят концепции и сложные сценарии в пошаговые инструкции, доступные даже начинающим.

В книге рассказывается о создании драйверов Windows. Однако речь идет не о работе с конкретным «железом», а о работе на уровне операционной системы (процессы, потоки, модули, реестр и многое другое).

Вы начнете с базовой информации о ядре и среде разработки драйверов, затем перейдете к API, узнаете, как создавать драйвера и клиентские приложения, освоите отладку, обработку запросов, прерываний и управление уведомлениями.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

Права на издание получены по соглашению с Pavel Yosifovich. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1977593375 англ.
ISBN 978-5-4461-1680-5

© 2019 by Pavel Yosifovich
© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021

Оглавление

Глава 1. Обзор внутреннего устройства Windows 11

Процессы 11

Виртуальная память 13

Состояние страниц 16

Системная память 17

Потоки 18

Стеки потоков 19

Системные сервисные функции 21

Общая архитектура системы 23

Дескрипторы и объекты 26

Имена объектов 27

Обращение к существующим объектам 30

Глава 2. Первые шаги в программировании для режима ядра 33

Установка инструментов 33

Создание проекта драйвера 34

Функция DriverEntry и функция выгрузки 36

Установка и загрузка драйвера 38

Простая трассировка 41

Упражнения 44

Итоги 44

Глава 3. Основы программирования ядра 45

Общие рекомендации программирования ядра 45

Необработанные исключения 46

Завершение 47

Возвращаемые значения функций 48

IRQL 48

Использование C++ 48

Тестирование и отладка 50
Отладочные и конечные сборки 50
API режима ядра 51
Функции и коды ошибок 52
Строки 53
Динамическое выделение памяти 55
Списки 57
Объект драйвера 59
Объекты устройств 60
Итоги 63

Глава 4. Драйвер: от начала до конца 64

Введение 64
Инициализация драйвера 66
Передача информации драйверу 68
Протокол обмена данными между клиентом и драйвером 69
Создание объекта устройства 70
Клиентский код 73
Функции диспетчеризации Create и Close 75
Функция диспетчеризации DeviceIoControl 76
Установка и тестирование 81
Итоги 83

Глава 5. Отладка 84

Средства отладки для Windows 84
Знакомство с WinDbg 86
Основы отладки пользовательского режима 87
Отладка режима ядра 102
Локальная отладка режима ядра 102
Знакомство с локальной отладкой режима ядра 104
Полная отладка режима ядра 110
Настройка управляемой машины 111
Настройка хоста 113
Основы отладки режима ядра 114
Итоги 118

Глава 6. Механизмы режима ядра 119
Уровень запроса прерывания 119
Повышение и понижение IRQL 123
Приоритеты потоков и IRQL 124
Отложенные вызовы процедур 124
Использование DPC с таймером 127
Асинхронные вызовы процедур 128
Критические секции и защищенные секции 129
Структурированная обработка исключений 130
Использование __try/__except 132
Использование __try/__finally 134
Использование RAII-оберток C++ вместо __try/__finally 135
Фатальный сбой 138
Информация дампа 140
Анализ файла дампа 144
Зависание системы 146
Синхронизация потоков 148
Операции со взаимоблокировкой 148
Объекты диспетчеризации 150
Мьютекс 153
Быстрый мьютекс 156
Семафор 157
Событие 158
Ресурс исполнительной системы 159
Синхронизация при высоких уровнях IRQL 160
Спин-блокировка 162
Рабочие элементы 166
Итоги 168

Глава 7. Пакеты запросов ввода/вывода (IRP) 169
Знакомство с IRP 169
Узлы устройств 170
Последовательность действий при работе с IRP 174
IRP и позиция стека ввода/вывода 176
Просмотр информации об IRP 179
Функции диспетчеризации 181
Завершение запроса 182

Обращение к пользовательским буферам . . . 184
Буферизованный ввод/вывод . . . 185
Прямой ввод/вывод . . . 189
Пользовательские буферы для запросов IRP_MJ_DEVICE_CONTROL . . . 193
Всё вместе: драйвер Zero . . . 195
Использование предварительно откомпилированного заголовка . . . 196
Функция DriverEntry . . . 198
Функция диспетчеризации для чтения . . . 199
Функция диспетчеризации для записи . . . 200
Тестовое приложение . . . 201
Итоги . . . 202

Глава 8. Уведомления потоков и процессов . . . 203

Уведомления процессов . . . 203
Реализация уведомлений процессов . . . 207
Функция DriverEntry . . . 209
Обработка уведомлений о выходе из процессов . . . 211
Обработка уведомлений о создании процессов . . . 213
Передача данных в пользовательский режим . . . 215
Клиент пользовательского режима . . . 217
Уведомления потоков . . . 220
Уведомления о загрузке образов . . . 222
Упражнения . . . 224
Итоги . . . 225

Глава 9. Уведомления объектов и реестра . . . 226

Уведомления объектов . . . 226
Обратный вызов перед операцией . . . 229
Обратный вызов после операции . . . 231
Драйвер Process Protector . . . 232
Регистрация уведомлений объектов . . . 233
Управление защищенными процессами . . . 234
Обратный вызов перед операцией . . . 238
Клиентское приложение . . . 238
Уведомления реестра . . . 241
Обработка уведомлений перед операцией . . . 243

Обработка уведомлений после операции . . . 243
Факторы быстродействия . . . 244
Реализация уведомлений реестра . . . 245
Обработка обратных вызовов уведомлений реестра . . . 246
Обновленный код клиента . . . 248
Упражнения . . . 250
Итоги . . . 250

Глава 10. Мини-фильтры файловой системы . . . 251

Введение . . . 252
Загрузка и выгрузка . . . 253
Инициализация . . . 255
Регистрация обратных вызовов . . . 259
Приоритет Altitude . . . 263
Установка . . . 265
INF-файлы . . . 266
Установка драйвера . . . 274
Обработка операций ввода/вывода . . . 274
Обратные вызовы перед операцией . . . 274
Обратные вызовы после операции . . . 277
Драйвер Delete Protector . . . 279
Обратные вызовы перед созданием . . . 281
Обработка информации перед операцией . . . 285
Небольшой рефакторинг . . . 288
Построение обобщенной версии драйвера . . . 291
Тестирование измененного драйвера . . . 295
Имена файлов . . . 297
Компоненты имени файла . . . 299
RAII-обертка FLT_FILE_NAME_INFORMATION . . . 301
Альтернативный драйвер Delete Protector . . . 303
Обработка информации перед созданием и назначением информации . . . 310
Тестирование драйвера . . . 312
Контексты . . . 312
Управление контекстами . . . 315
Инициирование запросов ввода/вывода . . . 316
Драйвер File Backup . . . 318

Обратный вызов после создания . . . 321
Обратный вызов перед записью . . . 325
Обратный вызов после освобождения . . . 332
Тестирование драйвера . . . 333
Восстановление из резервных копий . . . 333
Взаимодействие с пользовательским режимом . . . 335
Создание порта передачи данных . . . 336
Подключение из пользовательского режима . . . 337
Отправка и получение сообщений . . . 339
Расширенный драйвер File Backup . . . 340
Клиент пользовательского режима . . . 342
Отладка . . . 344
Упражнения . . . 347
Итоги . . . 348

Глава 11. Разное . . . 349

Цифровые подписи драйверов . . . 349
Driver Verifier . . . 353
Пример сеанса Driver Verifier . . . 358
Использование платформенного API . . . 362
Драйверы-фильтры . . . 364
Реализация драйвера-фильтра . . . 366
Присоединение фильтров . . . 368
Присоединение фильтров в произвольное время . . . 370
Деинициализация фильтра . . . 372
Подробнее о драйверах-фильтрах физических устройств . . . 373
Device Monitor . . . 375
Добавление устройства для фильтрации . . . 376
Удаление устройства-фильтра . . . 379
Инициализация и выгрузка . . . 380
Обработка запросов . . . 382
Тестирование драйвера . . . 385
Результаты запросов . . . 390
Перехват операций драйверов . . . 392
Библиотеки режима ядра . . . 394
Итоги . . . 396
От издательства . . . 396

Глава 1

Обзор внутреннего устройства Windows

Здесь описываются важнейшие концепции внутреннего устройства Windows. Некоторые аспекты будут более подробно описаны позднее в книге, когда они будут тесно связаны с непосредственно рассматриваемой темой. Убедитесь в том, что вы хорошо понимаете концепции этой главы, так как они образуют основу для построения любых драйверов и даже низкоуровневого кода пользовательского режима.

В этой главе:

- Процессы
- Виртуальная память
- Программные потоки
- Системные сервисные функции
- Архитектура системы
- Дескрипторы и объекты

Процессы

Процесс (process) — управляющий объект, который обеспечивает изоляцию адресных пространств и представляет работающий экземпляр программы. Довольно часто встречающееся выражение «процесс выполняется» неточно. Процессы не выполняются — они управляют. Потоки (threads) выполняют код и выполняются с технической точки зрения. На высоком уровне абстракции процессу принадлежит:

- Исполняемая программа, которая содержит код и данные, используемые для выполнения кода в процессе.

- Приватное виртуальное адресное пространство, которое используется для выделения памяти для любых целей, когда память потребуется коду внутри процесса.
- *Основной маркер* (primary token) — объект для хранения стандартного контекста безопасности процесса. Маркер используется потоками, выполняющими код внутри процесса (если только поток не переключится на использование другого маркера при помощи механизма олицетворения (impersonation)).
- Приватная таблица дескрипторов для объектов исполнительной системы (таких, как события, семафоры и файлы).
- Один или несколько потоков исполнения. Нормальный процесс пользовательского режима создается с одним потоком (в котором выполняется классическая функция `main`/`WinMain`). Процесс пользовательского режима без потоков в основном бесполезен, и в обычных обстоятельствах он будет уничтожен ядром.

Компоненты процесса изображены на рис. 1.1.

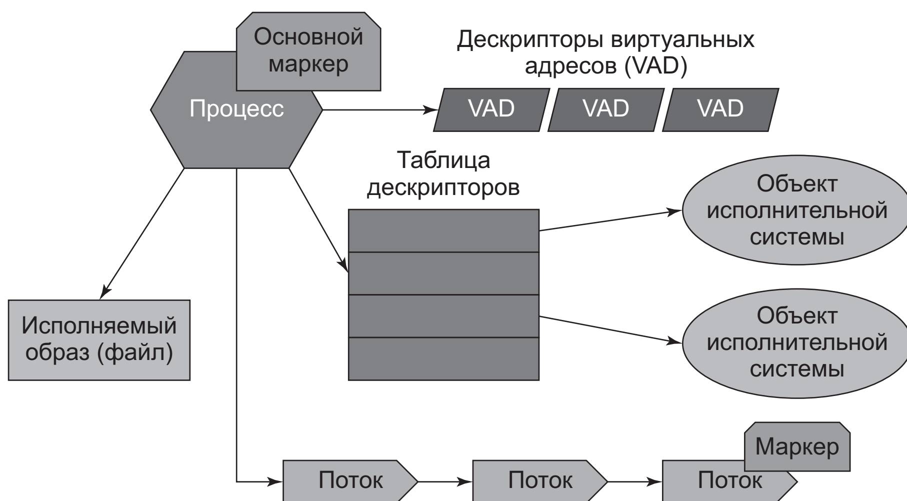


Рис. 1.1. Важнейшие компоненты процесса

Процесс однозначно определяется своим идентификатором процесса, который остается уникальным на все время существования объекта процесса в ядре. После уничтожения тот же идентификатор может повторно использоваться для новых процессов. Важно понимать, что сам исполняемый файл не обеспечивает однозначной идентификации процесса. Например, в системе могут одновременно работать пять экземпляров notepad.exe. Каждый процесс имеет

собственное адресное пространство, собственные потоки, собственную таблицу дескрипторов, собственный уникальный идентификатор процесса и т. д. Все пять процессов используют один и тот же файл образа (notepad.exe), в котором хранится их изначальный код и данные. На рис. 1.2 показан снимок экрана вкладки Details Диспетчера задач с пятью экземплярами Notepad.exe, каждый из которых обладает собственными атрибутами.

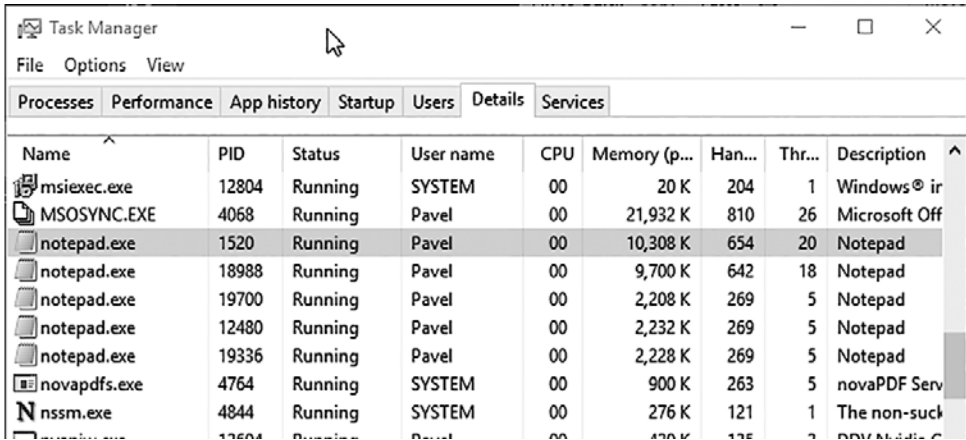


Рис. 1.2. Пять экземпляров notepad

Виртуальная память

Каждый процесс обладает собственным виртуальным приватным линейным адресным пространством. Это адресное пространство в исходном состоянии пусто (или почти пусто, потому что сначала в него отображается исполняемый образ и NtDll.Dll, а за ними следуют DLL-библиотеки других подсистем). Как только начинается выполнение основного (первого) потока, в адресном пространстве с большой вероятностью будет выделяться память, загружаться другие DLL-библиотеки и т. д. Это адресное пространство является приватным, то есть другие процессы не могут обращаться к нему напрямую. Диапазон адресов адресного пространства начинается с нуля (точнее, первые 64 Кбайт адресов не могут выделяться или использоваться иным образом) и следует до максимума, который зависит от разрядности (32 или 64 бита) процесса и разрядности операционной системы следующим образом:

- Для 32-разрядных процессов в 32-разрядных системах Windows размер адресного пространства процесса по умолчанию составляет 2 Гбайт.
- Для 32-разрядных процессов в 32-разрядных системах Windows, использующих параметр расширения пользовательского адресного пространства (флаг

LARGEADDRESSAWARE в заголовке Portable Executable), размер адресного пространства процесса может достигать 3 Гбайт (в зависимости от конкретного значения параметра). Чтобы получить доступ к расширенному диапазону адресного пространства, исполняемый файл, на основе которого был создан процесс, должен быть помечен флагом компоновщика LARGEADDRESSAWARE в заголовке. При отсутствии такого флага адресное пространство будет ограничено 2 Гбайт.

- Для 64-разрядных процессов (естественно, в 64-разрядных системах Windows) размер адресного пространства процесса составляет 8 Тбайт (Windows 8 и ранее) или 128 Тбайт (Windows 8.1 и далее).
- Для 32-разрядных процессов в 64-разрядных системах Windows размер адресного пространства составляет 4 Гбайт, если исполняемый файл был скомпонован с флагом LARGEADDRESSAWARE. В противном случае размер остается равным 2 Гбайт.

Требование флага LARGEADDRESSAWARE обусловлено тем фактом, что 2-гигабайтное адресное пространство требует только 31 разряда, а старший бит (MSB, Most Significant Bit) остается свободным и может использоваться приложением. Установка флага означает, что программа не использует разряд 31[1] ни для каких целей, поэтому присваивание 1 этому разряду (что происходит с адресами выше 2 Гбайт) не создаст проблем.

Каждый процесс имеет собственное адресное пространство, из-за чего все адреса в процессе относительны, а не абсолютны. Например, если вы пытаетесь определить, какие данные хранятся по адресу 0x20000, одного адреса недостаточно; необходимо указать, к какому процессу относится этот адрес.

Сама память называется *виртуальной*; это означает, что между диапазоном адресов и его точным расположением в физической памяти (ОЗУ) существует косвенная связь. Буфер в процессе может быть отображен в физическую память, а может временно храниться в файле (например, в страничном файле). Термин «виртуальный» относится к тому факту, что с точки зрения исполнения нет необходимости знать, хранятся ли данные, к которым вы обращаетесь, в оперативной памяти или нет. Если память процесса действительно отображена в оперативную память, то процессор обратится к данным напрямую. В противном случае процессор инициирует исключение ошибки страницы (page fault), что заставляет обработчика ошибок страниц диспетчера памяти загрузить данные из соответствующего файла, скопировать их в оперативную память, внести необходимые изменения в элементы таблицы страниц, обеспе-

[1] Имеется в виду, что это не 31-й разряд, а 32-й (нумерация начинается с 0-го). — *Примеч. пер.*

чивающие отображение буфера, и приказать процессору повторить попытку. На рис. 1.3 показано отображение виртуальной памяти в физическую для двух процессов.

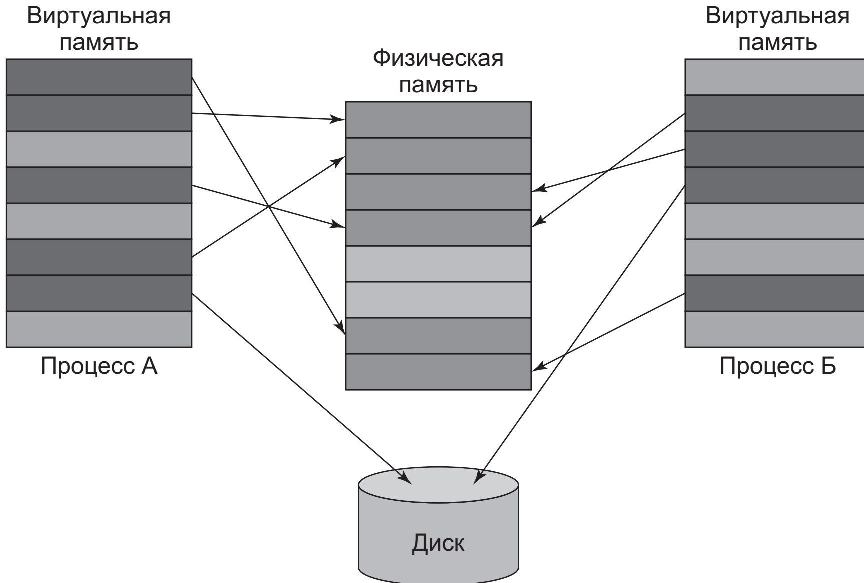


Рис. 1.3. Отображение виртуальной памяти

Единица управления памятью называется *страницей* (page). Каждый атрибут, относящийся к памяти (например, защита), всегда имеет страничную гранулярность. Размер страницы определяется типом процессора (и на некоторых процессорах может настраиваться); в любом случае диспетчер памяти должен использовать именно этот размер. Нормальный размер страницы (иногда называемый малым) — 4 Кбайт во всех архитектурах, поддерживаемых Windows.

Помимо нормального (малого) размера страницы, в системах Windows также поддерживаются большие страницы. Размер большой страницы равен 2 Мбайт (x86/x64/ARM64) и 4 Мбайт (ARM). Для отображения большой страницы без использования таблицы страниц используется элемент PDE (Page Directory Entry). Этот механизм ускоряет преобразование, но что еще важнее, он позволяет более эффективно использовать TLB (Translation Lookaside Buffer) — кэш недавно преобразованных страниц, содержимое которого поддерживается процессором. В случае большой страницы один элемент TLB позволяет отображать объем памяти, значительно превышающий размер малой страницы.

Недостаток больших страниц — необходимость наличия непрерывного участка физической памяти. Такого участка может не быть, если памяти в системе недостаточно или она сильно фрагментирована. Кроме того, большие страницы всегда невыгружаемые (non-pageable), и они должны быть защищены доступом только для чтения/записи. В Windows 10 и Server 2016 поддерживаются огромные страницы с размером 1 Гбайт. Они используются автоматически с большими страницами, если размер выделяемого блока превышает 1 Гбайт, а страница может быть выделена в непрерывном диапазоне физической памяти.

Состояние страниц

Каждая страница в виртуальной памяти может находиться в одном из трех состояний:

- *Свободная* (free) — память никак не используется; в ней нет ничего полезного. Любая попытка обращения к свободной странице приведет к исключению нарушения прав доступа. Большинство страниц только что созданного процесса свободно.
- *Закрепленная* (committed) — состояние, обратное свободному; к закрепленной странице возможно успешное обращение (без учета атрибутов защиты — например, попытка записи в страницу, доступную только для чтения, приведет к нарушению прав доступа). Закрепленные страницы обычно отображаются в физическую память или в файл (например, в страничный файл).
- *Зарезервированная* (reserved) — страница не закреплена, но диапазон адресов зарезервирован для возможного выделения в будущем. С точки зрения процессора зарезервированная страница не отличается от свободной — при любой попытке обращения происходит исключение нарушения прав доступа. Тем не менее новые попытки выделения памяти функцией `VirtualAlloc` (или `NtAllocateVirtualMemory`, соответствующей платформенной функцией) без указания конкретного адреса не приведут к выделению памяти в зарезервированном блоке. Классический пример использования зарезервированной памяти для поддержания непрерывного виртуального адресного пространства с экономией памяти описан позднее в этой главе, в разделе «Стеки потоков».

Системная память

Нижняя часть адресного пространства предназначена для использования процессами. Пока некоторый поток выполняется, связанное с ним адресное пространство процесса становится видимым от нулевого адреса до верхнего предела, описанного в предыдущем разделе. Однако операционная система тоже должна где-то находиться, а именно в верхнем диапазоне адресов, поддерживаемом системой:

- В 32-разрядных системах, работающих без параметра расширения пользовательского виртуального адресного пространства, операционная система размещается в верхних 2 гигабайтах виртуального адресного пространства, в диапазоне адресов от 0x8000000 до 0xFFFFFFFF.
- В 32-разрядных системах, настроенных в режиме расширения пользовательского виртуального адресного пространства, операционная система размещается в оставшемся адресном пространстве. Например, если система настроена с 3 Гбайт пользовательского адресного пространства на процесс (максимум), то ОС занимает верхний 1 Гбайт (в диапазоне адресов от 0xC0000000 до 0xFFFFFFFF). От сокращения адресного пространства в наибольшей степени страдает кэш файловой системы.
- В 64-разрядных системах Windows 8, Server 2012 и ранее ОС занимает верхние 8 Тбайт виртуального адресного пространства.
- В 64-разрядных системах Windows 8.1, Server 2012 R2 и далее ОС занимает верхние 128 Тбайт виртуального адресного пространства.

Системное пространство не является относительным по отношению к процессам — в конце концов, все процессы в системе используют одну и ту же «систему», одно ядро и одни драйверы (исключение — часть системной памяти, существующая на уровне сеанса, но для данного обсуждения это несущественно). Отсюда следует, что любой адрес в системном пространстве является абсолютным, а не относительным, потому что он «выглядит» одинаково в контексте каждого процесса. Конечно, попытки обращения из пользовательского режима в системное пространство приводят к исключению нарушения прав доступа.

В системном пространстве находится само ядро, уровень абстрагирования оборудования (HAL, Hardware Abstraction Layer) и загруженные драйверы ядра. Таким образом, драйверы ядра автоматически защищены от прямых обращений из пользовательского режима. Также это означает, что их влияние распространяется на всю систему. Например, если в драйвере ядра происходит утечка памяти, эта память не будет освобождена даже после выгрузки драйверов. С другой стороны, любая утечка в процессах пользовательского

режима никогда не может продолжаться за пределами их жизненного срока. Ядро отвечает за закрытие и освобождение всех ресурсов, приватных для уничтожаемого процесса (все дескрипторы закрываются, а вся приватная память освобождается).

Потоки

Фактическое выполнение кода осуществляется *потоками* (threads). Поток содержится в процессе и использует ресурсы, предоставляемые процессом (например, виртуальную память и дескрипторы объектов ядра), для выполнения работы.

Самая важная информация, принадлежащая потоку:

- Текущий режим доступа (пользовательский режим или режим ядра).
- Контекст выполнения, включающий значения регистров процессора и состояние выполнения.
- Один или два стека, используемые для выделения памяти локальных переменных и управления вызовами.
- Массив локальной памяти потоков (TLS, Thread Local Storage), предоставляющий средства для хранения приватных данных потока с унифицированной семантикой доступа.
- Базовый приоритет и текущий (динамический) приоритет.
- Привязка к процессору, указывающая, на каких процессорах разрешено выполнение потока.

Наиболее распространенные состояния, в которых может находиться поток:

- Выполнение (Running) — поток выполняет код на (логическом) процессоре.
- Готовность (Ready) — поток ожидает планирования на выполнение, потому что все нужные процессоры либо заняты, либо недоступны.
- Ожидание (Waiting) — поток ожидает наступления некоторого события, прежде чем продолжить выполнение. После того как событие произойдет, поток переходит в состояние готовности.

На рис. 1.4 изображена соответствующая диаграмма состояний. Числа в круглых скобках обозначают номера состояний при просмотре в таких программах, как Performance Monitor. Обратите внимание: у состояния готовности (Ready) имеется парное состояние отложенной готовности (Deferred Ready), сходное с ним и существующее в основном для минимизации внутренних блокировок.

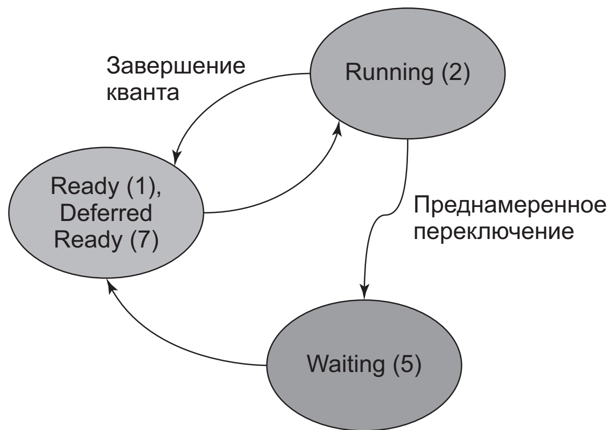


Рис. 1.4. Основные состояния потоков

Стеки потоков

У каждого потока имеется стек, используемый им при выполнении. Стек используется для создания локальных переменных, передачи параметров функциям (в некоторых случаях) и хранения адресов возврата при вызове функций. У потока имеется как минимум один стек, находящийся в системном пространстве (пространстве ядра); он относительно мал (по умолчанию 12 Кбайт в 32-разрядных системах и 24 Кбайт в 64-разрядных системах). У потоков пользовательского режима существует второй стек в диапазоне адресов пользовательского режима соответствующего процесса, этот стек имеет намного больший размер (по умолчанию он может увеличиваться до 1 Мбайт). На рис. 1.5 изображен пример с тремя потоками пользовательского режима и их стеками. На рисунке потоки 1 и 2 принадлежат процессу А, а поток 3 — процессу Б.

Стек ядра всегда находится в физической памяти, пока поток находится в состоянии выполнения или готовности. Причина будет рассмотрена позднее в этой главе. С другой стороны, стек пользовательского режима может выгружаться, как и все содержимое памяти пользовательского режима.

Поведение стека пользовательского режима отличается от стека режима ядра из-за своего размера. Он начинается с закрепления небольшого объема памяти (вплоть до одной страницы), при этом остаток адресного пространства стека составляет зарезервированная память (которая не может выделяться ни для каких целей). Идея состоит в том, чтобы иметь возможность расширять стек в том случае, если коду потока потребуется использовать больший объем памяти стека. Для этого следующая страница (иногда несколько страниц) не-

посредственно после закрепленной части помечается специальным защитным атрибутом PAGE_GUARD — признаком сторожевой страницы. Если потоку понадобится больше памяти, он выполняет запись в сторожевую страницу; возникает исключение, которое обрабатывается диспетчером памяти. Затем диспетчер памяти снимает атрибут сторожевой страницы, закрепляет страницу и помечает следующую страницу как сторожевую. Таким образом, стек растет по мере надобности, а вся память стека не закрепляется заранее. На рис. 1.6 изображен примерный вид стека потока пользовательского режима.

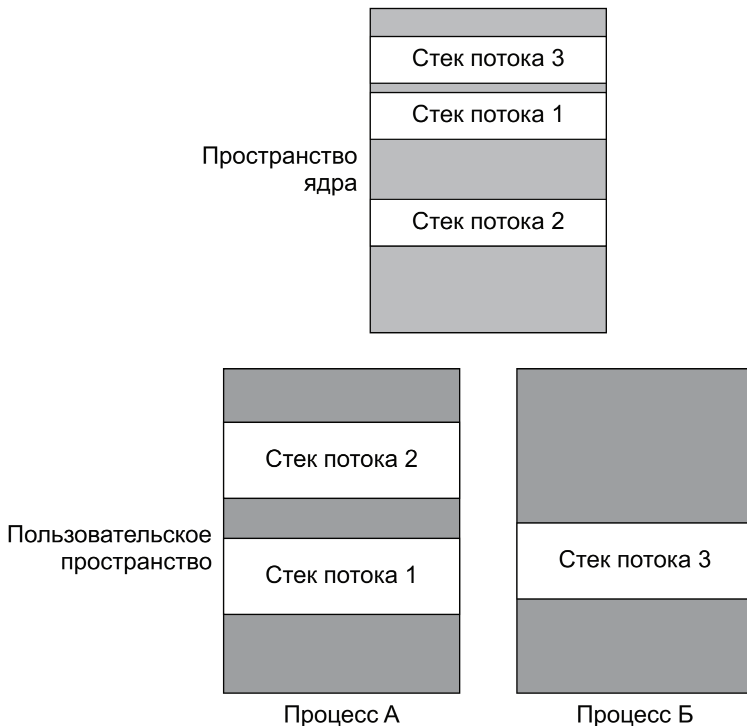


Рис. 1.5. Потоки пользовательского режима и их стеки

Размер стека пользовательского режима для потока определяется следующим образом:

- Величины закрепленной и зарезервированной части стека хранятся в заголовке PE (Portable Executable) исполняемого файла. Они используются по умолчанию, если поток не укажет альтернативные значения.
- При создании потока функцией CreateThread (или аналогичной функцией) вызывающая сторона может указать требуемый размер стека — либо размер

изначально закрепленной части, либо размер зарезервированной части (но не оба сразу) в зависимости от флага, переданного функции; при передаче нуля используются значения по умолчанию из предыдущего пункта.

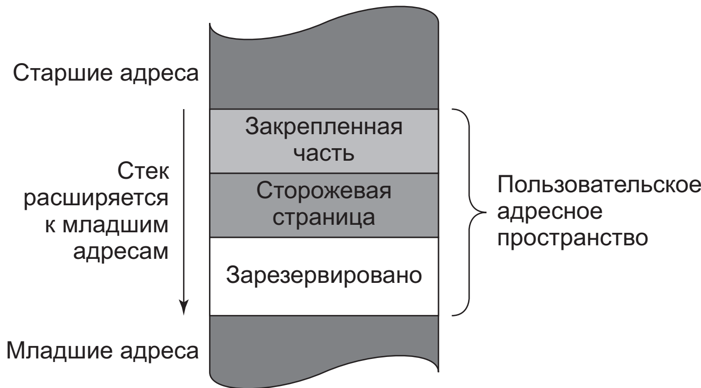


Рис. 1.6. Стек потока в пользовательском пространстве

Интересно, что функции `CreateThread` и `CreateRemoteThread(Ex)` позволяют задать только одно значение для размера стека и состояния памяти (закрепленная или зарезервированная). Платформенная (недокументированная) функция `NtCreateThreadEx` позволяет задать оба значения.

Системные сервисные функции

Приложениям требуется выполнять различные операции, которые не являются чисто вычислительными: выделение памяти, открытие файлов, создание потоков и т. д. Эти операции могут выполняться только кодом, работающим в режиме ядра. Как же выполнять такие операции в коде пользовательского режима? Возьмем классический пример: пользователь, запустивший процесс Notepad, выполняет запрос на открытие файла командой меню File. Код Notepad реагирует вызовом документированной функции Windows API `CreateFile`. Функция `CreateFile` документирована в соответствии с реализацией из kernel32.dll — одной из DLL-библиотек подсистем Windows. Эта функция также работает в пользовательском режиме, поэтому она ни при каких условиях не сможет напрямую открыть файл. После проверки ошибок она вызывает функцию `NtCreateFile`. Реализация этой функции содержится в NTDLL.dll — фундаментальной DLL-библиотеке, реализующей так называемый платформенный API; по сути, это код самого низкого уровня, который все еще работает в пользовательском режиме. Эта (официально недокументированная) функция API

осуществляет переход в режим ядра. Непосредственно перед переходом она помещает число, называемое *номером системной сервисной функции*, в регистр процессора (EAX в архитектурах Intel/AMD). Затем выполняется специальная команда процессора (`syscall` для x64, `sysenter` для x86), которая осуществляет фактический переход в режим ядра с переходом в заранее определенную функцию, называемую диспетчером системных функций.

В свою очередь, диспетчер системных функций использует значение из регистра `EAX` как индекс в таблице SSDT (System Service Dispatch Table). По адресу, содержащемуся в таблице, код осуществляет переход непосредственно к системной функции. Для нашего примера с Notepad элемент SSDT содержит указатель на функцию `NtCreateFile` диспетчера ввода/вывода. Обратите внимание: имя этой функции совпадает с именем функции из NTDLL.dll; более того, она получает те же аргументы. После того как вызов системной функции будет завершен, поток возвращается в пользовательский режим для выполнения команды, следующей за `sysenter`/`syscall`. Эта последовательность событий изображена на рис. 1.7.

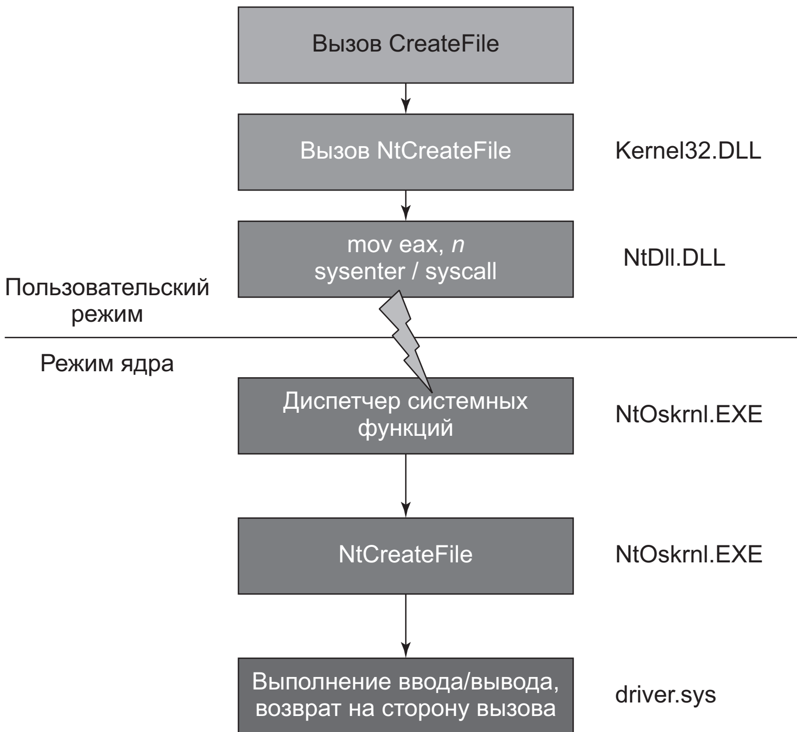


Рис. 1.7. Последовательность действий при вызове системных сервисных функций

Общая архитектура системы

На рис. 1.8 изображена общая архитектура Windows, состоящая из компонентов пользовательского режима и режима ядра.

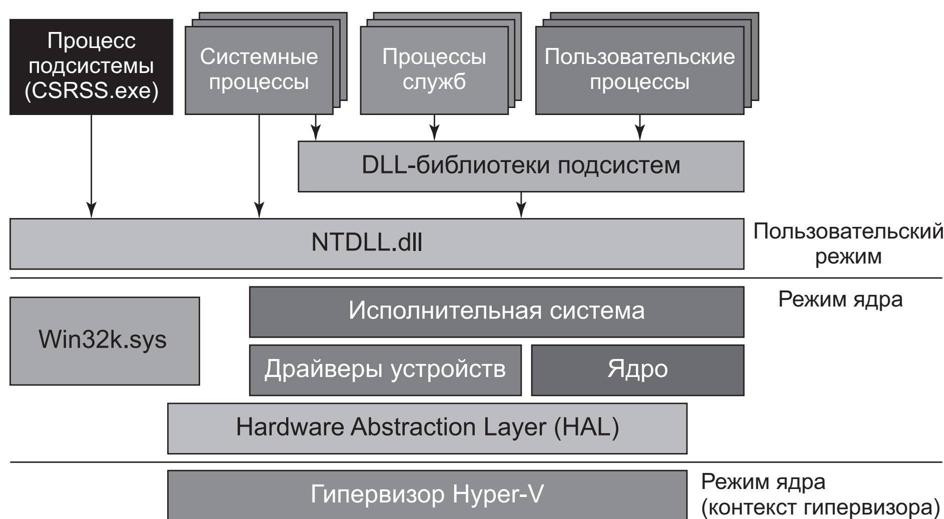


Рис. 1.8. Архитектура системы Windows

Краткая сводка блоков на рис. 1.8:

- **Пользовательские процессы**. Обычные процессы, созданные на базе файлов образов и выполняемые в системе (например, экземпляры Notepad.exe, cmd.exe, explorer.exe и т. д.).
- **DLL-библиотеки подсистем**. DLL-библиотеки подсистем представляют собой библиотеки динамической компоновки (DLL, Dynamic Link Libraries), реализующие API подсистем. Подсистема является неким представлением функциональности, предоставляемым ядром. С технической точки зрения, начиная с Windows 8.1 существует только одна подсистема — подсистема Windows. К числу DLL-библиотек подсистем принадлежат такие известные файлы, как kernel32.dll, user32.dll, gdi32.dll, advapi32.dll, combase.dll и др. В основном DLL-библиотеки содержат официально документированный Windows API.
- **NTDLL.DLL**. DLL-библиотека системного уровня, реализующая платформенный Windows API. Она содержит код самого низкого уровня, выполняемый в пользовательском режиме. Самая важная ее роль — переход в режим ядра для вызова системных функций. NTDLL также реализует диспетчер

кучи (Heap Manager), загрузчик образов (Image Loader) и некоторые части пула потоков пользовательского режима.

- **Процессы служб**. Процессы служб — нормальные процессы Windows, которые взаимодействуют с диспетчером служб (SCM, Service Control Manager — реализуется в services.exe) и позволяют до определенной степени управлять своим сроком жизни. Диспетчер служб может запускать, останавливать, приостанавливать, возобновлять работу служб и отправлять службам другие сообщения. Службы обычно выполняются под одной из специальных учетных записей Windows — локальной системы, сетевых или локальных служб.
- **Исполнительная система**. Исполнительная система является верхним уровнем NtOskrnl.exe («ядра»). В ней содержится большая часть кода, работающего в режиме ядра. Прежде всего это различные диспетчеры: диспетчер объектов, диспетчер памяти, диспетчер ввода/вывода, диспетчер Plug & Play, диспетчер электропитания, диспетчер конфигурации и т. д. По своим размерам она значительно больше нижнего уровня ядра.
- **Ядро**. Уровень ядра реализует самые фундаментальные и критичные по времени части кода ОС режима ядра. К их числу относятся планирование потоков, обработка прерываний и диспетчеризация исключений, а также реализация различных примитивов ядра, таких как мьютексы и семафоры. Часть кода ядра написана на машинном языке конкретного процессора для эффективности и для получения прямого доступа к специфическим возможностям процессора.
- **Драйверы устройств**. Драйверы устройств представляют собой загружаемые модули ядра. Их код выполняется в режиме ядра и может распоряжаться всей мощью ядра. Книга посвящена написанию некоторых разновидностей драйверов ядра.
- **Win32k.sys**. Компонент режима ядра подсистемы Windows. По сути, это модуль ядра (драйвер), который обеспечивает часть Windows API и классического интерфейса графических устройств GDI (Graphic Device Interface), относящуюся к пользовательскому интерфейсу. Это означает, что все операции оконной системы (`CreateWindowEx`, `GetMessage`, `PostMessage` и т. д.) обеспечиваются этим компонентом. Остальные компоненты системы практически ничего не знают о пользовательском интерфейсе.
- **Уровень абстрагирования оборудования (HAL)**. Уровень HAL располагается над оборудованием в максимальной близости к процессору. Он позволяет драйверам устройств использовать API, не требующие досконального и точного знания всех подробностей, например контроллер прерываний или контроллер DMA. Естественно, этот уровень в основном представляет интерес для драйверов, написанных для управления физическими устройствами.

В Windows 10 версии 1607 появилась поддержка подсистемы Windows для Linux (WSL, Windows Subsystem for Linux). И хотя на первый взгляд это всего лишь очередная подсистема вроде старых подсистем POSIX и OS/2, поддерживаемых Windows, на самом деле это совсем не так. Старые подсистемы могли выполнять приложения POSIX и OS/2, если они были откомпилированы компилятором для Windows. С другой стороны, в WSL такое требование отсутствует. Существующие исполняемые файлы для Linux (в формате ELF) могут запускаться в Windows без перекомпиляции.

Чтобы такая схема работала, был создан новый тип процессов — процесс Pico в сочетании с провайдером Pico. Вкратце процесс Pico представляет собой пустое адресное пространство (минимальный процесс), используемое для процессов WSL, где каждый системный вызов (вызов системной функции Linux) должен быть перехвачен и преобразован в эквивалентный вызов(-ы) системной функции Windows при помощи провайдера Pico. Таким образом, на Windows-машине фактически устанавливается полноценная система Linux (часть пользовательского режима).

- **Системные процессы**. Общим термином «системные процессы» обозначаются процессы, которые обычно просто «находятся на своем месте» и делают то, что положено; обычно эти процессы не предназначены для прямого взаимодействия. Тем не менее они важны, а некоторые даже необходимы для благополучного существования системы. Завершение некоторых из этих процессов приводит к фатальным последствиям вплоть до полного сбоя системы. Некоторые из системных процессов относятся к платформенным; это означает, что они используют только платформенный API (API, реализуемый NTDLL). Примеры системных процессов — Smss.exe, Lsass.exe, Winlogon.exe, Services.exe и др.
- **Процесс подсистемы**. Процесс подсистемы Windows, в котором выполняется образ Csrss.exe, может рассматриваться как помощник ядра для управления процессами, работающими в системе Windows. Этот процесс является критическим, то есть в случае его уничтожения происходит полный сбой системы. Обычно создается только один экземпляр Csrss.exe для каждого сеанса, поэтому в стандартной системе существуют два экземпляра — для сеанса 0 и для сеанса текущего пользователя (обычно 1). Хотя Csrss.exe является «диспетчером» подсистемы Windows (единственным из оставшихся в наши дни), его важность выходит далеко за рамки этой роли.
- **Гипервизор Hyper-V**. Гипервизор Hyper-V существует в Windows 10 и Server 2016 (и последующих системах), если они поддерживают механизм VBS (Virtualization Based Security). VBS предоставляет дополнительный уровень безопасности, где реальная машина в действительности представлена виртуальной машиной, находящейся под управлением Hyper-V. Рассмотрение

VBS выходит за рамки книги. За дополнительной информацией обращайтесь к книге «Внутреннее устройство Windows»[1].

Дескрипторы и объекты

Ядро Windows предоставляет различные типы объектов, которые могут использоваться процессами пользовательского режима, самим ядром и драйверами режима ядра. Экземпляры этих типов представляют собой структуры данных в системном пространстве, создаваемые диспетчером объектов (часть исполнительной системы) по требованию кода пользовательского режима или режима ядра. Для объектов ведется подсчет ссылок — только после освобождения последней ссылки объект будет уничтожен и удален из памяти.

Так как экземпляры объектов находятся в системном пространстве, код пользовательского режима не может обращаться к ним напрямую. Он должен использовать механизм косвенного обращения — так называемые дескрипторы (handles). Дескриптор представляет собой индекс в таблице, хранимой на уровне отдельных процессов, которая содержит логические указатели на объект ядра, находящийся в системном пространстве. Существуют различные функции `Create*` и `Open*` для создания/открытия объектов и получения дескрипторов этих объектов. Например, функция пользовательского режима `CreateMutex` позволяет создать или открыть мьютекс (в зависимости от того, задано ли имя объекта и существует ли он). В случае успеха функция возвращает дескриптор этого объекта. Возвращаемое значение 0 является признаком недействительного дескриптора (и неудачного вызова функции). С другой стороны, функция `OpenMutex` пытается открыть дескриптор для именованного мьютекса. Если объект с заданным именем не существует, вызов функции завершается неудачей, и функция возвращает `null` (0).

Код режима ядра (и драйверов) может использовать как дескриптор, так и прямой указатель на объект. Выбор обычно зависит от функции API, которую код хочет вызвать. В некоторых случаях дескриптор, передаваемый драйверу из пользовательского режима, должен быть преобразован в указатель функцией `ObReferenceObjectByHandle`. Эти подробности будут рассмотрены в одной из следующих глав.

Значения дескрипторов кратны 4, при этом первый допустимый дескриптор равен 4; нулевое значение дескриптора ни при каких условиях действительным быть не может.

[1] *Руссинович М., Соломон Д., Ионеску А., Йосифович П.* Внутреннее устройство Windows. 7-е изд. — СПб.: Питер, 2019. — 944 с.: ил.

Многие функции возвращают `null (0)` при неудаче, но есть и исключения. В первую очередь функция `CreateFile` в случае неудачи возвращает `INVALID_HANDLE_VALUE (-1)`.

Код режима ядра может использовать дескрипторы при создании/открытии объектов, но он также может использовать прямые указатели на объекты ядра. Обычно это делается в тех случаях, когда того требует определенная функция API. Код ядра может получить указатель на объект по действительному дескриптору при помощи функции `ObReferenceObjectByHandle`. В случае успеха счетчик ссылок объекта увеличивается; это делается для предотвращения риска того, что клиент пользовательского режима, удерживающий дескриптор, решит закрыть его. В этом случае указатель на объект, хранящийся у кода ядра, будет указывать на несуществующий объект (так называемый висячий указатель). К объекту можно безопасно обращаться независимо от того, где именно он удерживается, пока код ядра не вызовет функцию `ObDerefenceObject`, которая уменьшает счетчик ссылок. Если код ядра пропустит этот вызов, в системе возникнет утечка ресурсов, которая будет устранена только при следующей загрузке системы.

Для всех объектов ведутся счетчики ссылок. Диспетчер объектов хранит для объектов количество дескрипторов и общий счетчик ссылок. После того, как объект станет ненужным, его клиент должен закрыть дескриптор (если он использовался для обращения к объекту) или разыменовать объект (если клиент режима ядра использовал указатель). В дальнейшем код должен считать свой дескриптор/указатель недействительным. Диспетчер объектов уничтожает объект в том случае, если его счетчик ссылок упал до нуля.

Каждый объект содержит указатель на тип объекта, в котором хранится информация о самом типе; это означает, что для каждой разновидности объектов существует один объект типа. Они также предоставляются в форме экспортируемых глобальных переменных ядра; некоторые из них определяются в заголовках ядра и могут пригодиться в определенных ситуациях, как будет показано в следующих главах.

Имена объектов

Некоторые разновидности объектов могут обладать именами. Зная имя объекта, вы можете открыть объект по имени соответствующей функцией `Open`. Обратите внимание: не все объекты обладают именами; например, у процессов и потоков имен нет — есть только идентификаторы. Именно по этой причине функции `OpenProcess` и `OpenThread` должен передаваться идентификатор процесса/потока

(числа) вместо строкового имени. Другой довольно странный пример объекта, не обладающего именем, — файл. Имя файла не является именем объекта — это совершенно разные концепции.

В коде пользовательского режима вызов функции `Create` с передачей имени создает объект с этим именем, если такой объект не существует, но если объект существует, то функция просто открывает существующий объект. В последнем случае вызов `GetLastError` вернет значение `ERROR_ALREADY_EXISTS`, которое означает, что новый объект не создается, а функция возвращает просто еще один дескриптор для существующего объекта.

Имя, передаваемое функции `Create`, на самом деле не является окончательным именем объекта. К нему присоединяется префикс `\Sessions\`*x*`\BaseNamedObjects\`, где *x* — идентификатор сеанса вызывающей стороны. Если идентификатор сеанса равен 0, то к имени присоединяется префикс `\BaseNamedObjects\`. Если же вызывающая сторона работает в контейнере AppContainer (обычно это процесс Universal Windows Platform), то присоединяемая строка становится более сложной и содержит уникальный идентификатор SID контейнера `\Sessions\`*x*`\AppContainerNamedObjects\{`*AppContainerSID*`}`.

Из всего сказанного следует, что имена объектов относительны по отношению к сеансу (а в случае AppContainer — относительны по отношению к пакету). Если объект должен совместно использоваться разными сеансами, он может быть создан в сеансе 0, для чего к имени объекта присоединяется `Global\`; например, при создании функцией `CreateMutex` мьютекса с именем `Global\MyMutex` объект будет создан в иерархии`\BaseNamedObjects`. Следует заметить, что контейнеры AppContainer не обладают полномочиями для использования пространства имен объектов сеанса 0. Для просмотра этой иерархии можно воспользоваться программой WinObj из пакета Sysinternals (должна запускаться с повышенными привилегиями), как показано на рис. 1.9.

В окне на рис. 1.9 показано пространство имен диспетчера объекта, которое образует иерархию именованных объектов. Эта структура хранится в памяти, а для манипуляций с ней по мере надобности используется диспетчер объектов (часть исполнительной среды). Область видимости: безымянные объекты не входят в эту структуру; это означает, что в WinObj отображаются не все существующие объекты, а только объекты, созданные с указанием имени.

Каждый процесс содержит приватную таблицу дескрипторов объектов ядра (как именованных, так и безымянных). Для просмотра таблицы можно воспользоваться программами Process Explorer и/или Handles из пакета Sysinternals. На рис. 1.10 показан снимок экрана Process Explorer с дескрипторами некоторого процесса. По умолчанию в окне для каждого дескриптора выводится только тип объекта и имя. При этом также доступны другие столбцы, показанные на рис. 1.10.

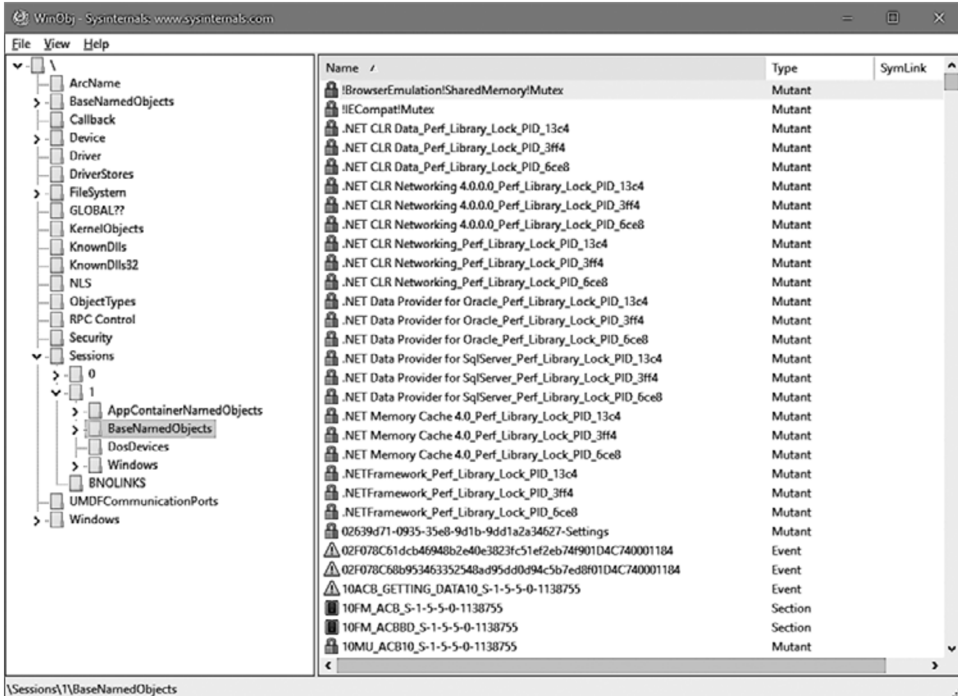


Рис. 1.9. Программа WinObj из пакета Sysinternals

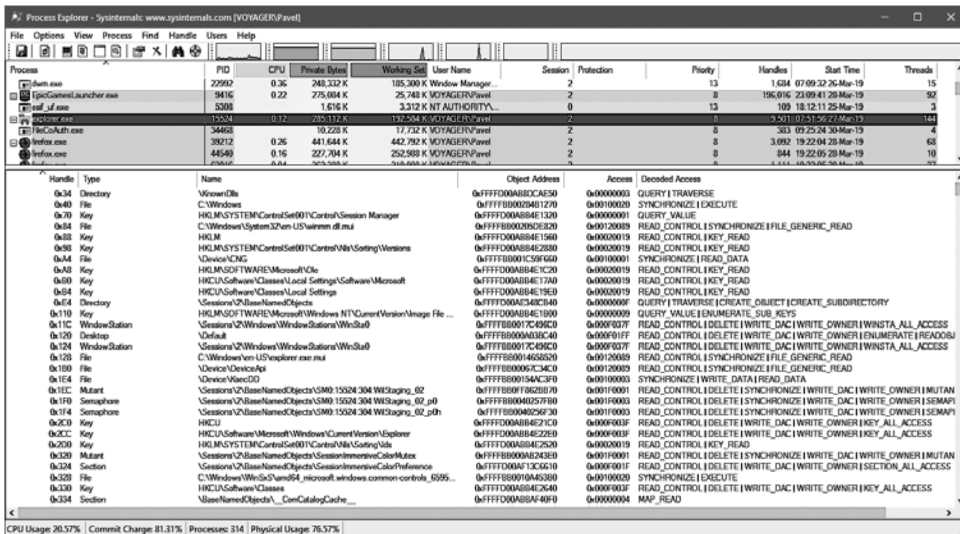


Рис. 1.10. Просмотр дескрипторов процессов в программе Process Explorer

По умолчанию Process Explorer отображает только дескрипторы для объектов с именами (в соответствии с определением имен в Process Explorer — см. далее). Чтобы просмотреть все дескрипторы в процессе, выберите команду Show Unnamed Handles and Mappings в меню View программы Process Explorer.

Различные столбцы в режиме вывода дескрипторов предоставляют дополнительную информацию о каждом дескрипторе. Значение дескриптора и тип объекта не нуждаются в пояснениях. С именем столбца дело обстоит сложнее. В нем приводятся истинные имена объектов для мьютексов (`Mutant`), семафоров (`Semaphore`), событий (`Event`), секций (`Section`), портов ALPC (`ALPC Port`), заданий (`Job`), таймеров (`Timer`) и других, не столь часто используемых типов. Для других объектов выводится имя, смысл которого отличается от истинного:

- Для объектов процессов (`Process`) и потоков (`Thread`) выводится уникальный идентификатор.
- Для объектов файлов (`File`) выводится имя файла (или имя устройства), на который указывает объект. Это имя не совпадает с именем объекта, так как по имени файла невозможно получить дескриптор для объекта файла — можно только создать новый объект файла, который соответствует тому же файлу или устройству (при условии, что это позволяют сделать настройки совместного доступа для исходного объекта файла).
- Для имен объектов разделов реестра (`Key`) выводится путь к разделу реестра. Имя раздела не может использоваться по тем же причинам, что и для объектов файлов.
- Для объектов каталогов (`Directory`) выводится путь вместо истинного имени объекта. Каталог не является объектом файловой системы, это каталоги диспетчера объектов — для их просмотра удобно использовать программу WinObj из пакета Sysinternals.
- Для имен объектов маркеров (`Token`) выводится имя пользователя, хранящееся в маркере.

Обращение к существующим объектам

В столбце Access в окне режима дескрипторов программы Process Explorer выводится маска доступа, использованная для создания или открытия дескриптора. Маска доступа определяет, какие операции могут выполняться с конкретным дескриптором. Например, если код клиента хочет завершить процесс, он должен сначала вызвать функцию `OpenProcess`, чтобы получить дескриптор нужного процесса с маской доступа (как минимум) `PROCESS_TERMINATE`; в противном случае завершить процесс с этим дескриптором не удастся. Если вызов завершится успехом, то вызов `TerminateProcess` тоже должен быть успешным.

Пример кода пользовательского режима для завершения процесса по идентификатору процесса:

```
bool KillProcess(DWORD pid) {
    // Открыть для процесса дескриптор с достаточным уровнем

    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);
    if (!hProcess)
        return false;

    // Уничтожить с произвольным кодом завершения
    BOOL success = TerminateProcess(hProcess, 1);

    // Закрыть дескриптор
    CloseHandle(hProcess);

    return success != FALSE;
}
```

В столбце Decoded Access приводится текстовое описание маски доступа (для некоторых типов объектов), по которому проще определить уровень доступа, разрешенный для конкретного дескриптора.

Если сделать двойной щелчок на элементе дескриптора, программа выводит некоторые свойства объекта. На рис. 1.11 показан снимок экрана со свойствами объекта события.

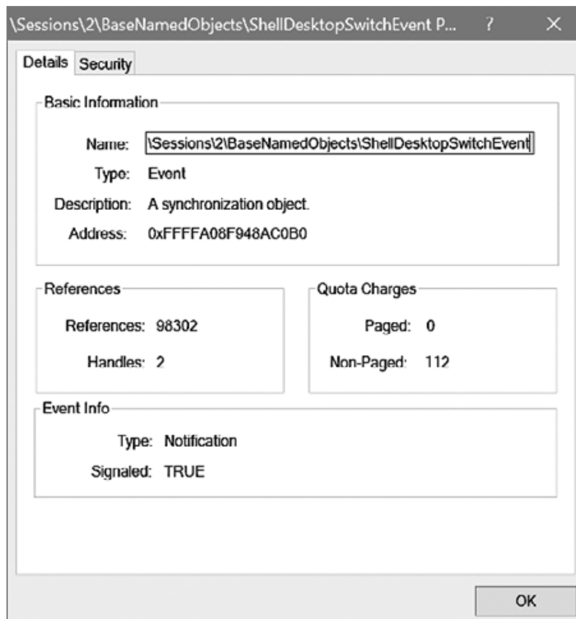


Рис. 1.11. Свойства объекта в Process Explorer

Свойства на рис. 1.11 включают имя объекта (если есть), его тип, описание, адрес в памяти ядра, количество открытых дескрипторов и информацию об объекте, например состояние и тип для объекта события. Учтите, что в поле References не выводится фактическое количество ссылок на объект. Для просмотра реального значения счетчика ссылок объекта используется команда `!trueref` отладчика ядра:

```
lkd> !object 0xFFFFA08F948AC0B0
Object: ffffa08f948ac0b0 Type: (ffffa08f684df140) Event
    ObjectHeader: ffffa08f948ac080 (new version)
    HandleCount: 2 PointerCount: 65535
    Directory Object: ffff90839b63a700 Name: ShellDesktopSwitchEvent
lkd> !trueref ffffa08f948ac0b0
ffffa08f948ac0b0: HandleCount: 2 PointerCount: 65535 RealPointerCount: 3
```

Атрибуты объектов и отладчик ядра более подробно рассматриваются в следующих главах.

А теперь займемся написанием очень простого драйвера для демонстрации многих инструментов, которые понадобятся нам позднее.

Глава 2

Первые шаги в программировании для режима ядра

В этой главе рассматриваются основы, необходимые для того, чтобы приступить к разработке драйвера режима ядра. В этой главе мы установим необходимые инструменты и напишем очень простой драйвер, который можно будет загружать и выгружать.

В этой главе:

- Установка инструментов
- Создание проекта драйвера
- Функция `DriverEntry` и функция выгрузки
- Развертывание драйвера
- Простая трассировка

Установка инструментов

В прежние времена (до 2012 года) в процессе разработки и построения драйверов приходилось применять специальные средства построения из пакета DDK (Device Driver Kit), без удобных интегрированных сред, к которым привыкли разработчики при написании приложений пользовательского режима. Существовали некоторые обходные решения, но все они были не идеальны и не поддерживались официально. К счастью, начиная с Visual Studio 2012 и Windows Driver Kit 8, компания Microsoft официально поддерживает построение драйверов в Visual Studio (и msbuild) без необходимости использовать отдельный компилятор и средства сборки.

Чтобы приступить к разработке драйвера, необходимо установить следующие инструменты (в указанном порядке):

- Visual Studio 2017 или 2019 с новейшими обновлениями. Проследите за тем, чтобы во время установки была выбрана поддержка C++. На момент написания книги среда Visual Studio 2019 была только что выпущена и могла использоваться для разработки драйверов. Вам подойдет любое издание, включая бесплатное издание Community edition.
- Windows 10 SDK (обычно новейшая версия является самой лучшей). Проследите за тем, чтобы во время установки был как минимум выбран пункт Debugging Tools for Windows.
- Windows 10 Driver Kit (WDK). Новейшая версия должна вам подойти, но также проследите за тем, чтобы в конце стандартной установки были установлены шаблоны проектов для Visual Studio.
- Пакет Sysinternals, оказывающий неоценимую помощь при любой работе с «внутренностями» системы, можно бесплатно загрузить по адресу *http://www.sysinternals.com*. Щелкните на ссылке Sysinternals Suite в левой части веб-страницы и загрузите Sysinternals в виде zip-архива. Распакуйте архив в любую папку; инструменты готовы к работе.

Чтобы убедиться в том, что шаблоны WDK были установлены правильно, откройте Visual Studio, выберите вариант New Project и проверьте наличие проектов драйверов — например, «Empty WDM Driver».

Создание проекта драйвера

После установки всех перечисленных средств можно создать новый проект драйвера. В этом разделе вам понадобится шаблон пустого драйвера WDM (WDM Empty Driver). На рис. 2.1 показано, как выглядит диалоговое окно New Project для этого типа драйверов в Visual Studio 2017. На рис. 2.2 изображена та же программа-мастер в Visual Studio 2019. На обеих иллюстрациях проекту присвоено имя «Sample».

После того как проект будет создан, на панели Solution Explorer находится всего один файл — Sample.inf. В данном примере этот файл не нужен, просто удалите его.

В проект нужно добавить исходный файл. Щелкните правой кнопкой мыши на узле Source Files на панели Solution Explorer и выберите команду Add / New Item... из меню File. Выберите исходный файл C++ и присвойте ему имя Sample.cpp. Щелкните на кнопке OK, чтобы создать файл.

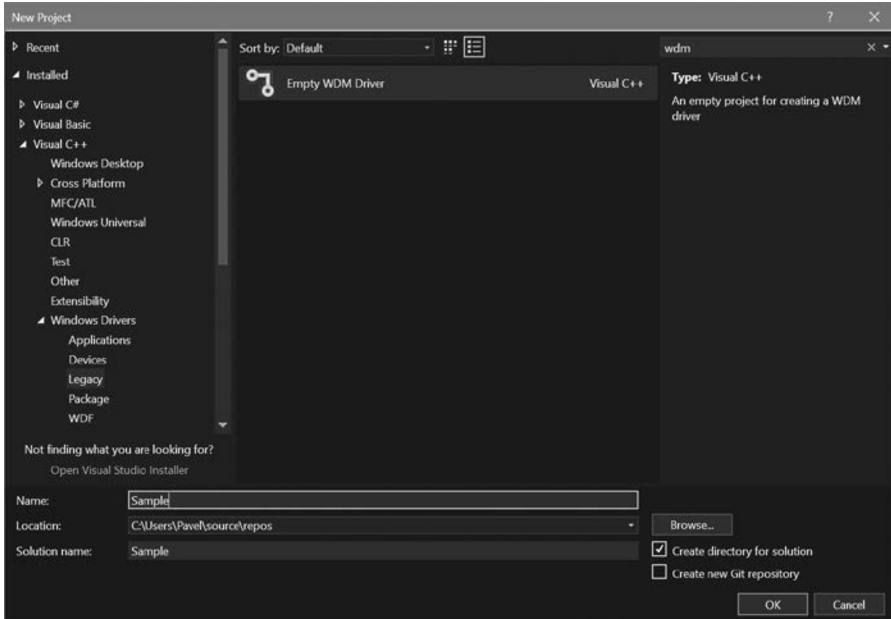


Рис. 2.1. Новый проект драйвера WDM в Visual Studio 2017

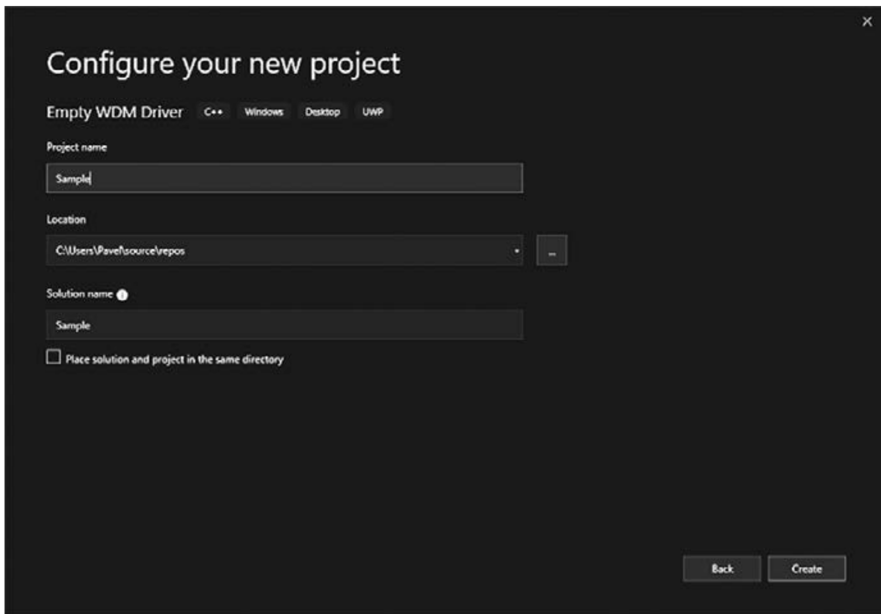


Рис. 2.2. Новый проект драйвера WDM в Visual Studio 2019

Функция DriverEntry и функция выгрузки

Каждый драйвер содержит точку входа, которой по умолчанию присваивается имя `DriverEntry`. Это своего рода аналог классической функции `main` приложений пользовательского режима. Эта функция вызывается системным потоком на уровне `IRQL PASSIVE_LEVEL` (0). (IRQL подробно рассматриваются в главе 8).

Функция `DriverEntry` имеет заранее определенный прототип:

```
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath);
```

Аннотации `_In_` являются частью языка аннотаций исходного кода SAL (Source (Code) Annotation Language). Эти аннотации, прозрачные для компилятора, предоставляют метаданные для читателя-человека и средств статического анализа кода. Мы будем использовать их по мере возможности, чтобы сделать код более понятным.

Минимальная функция `DriverEntry` может просто вернуть успешный код статуса:

```
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath)
{
    return STATUS_SUCCESS;
}
```

Такой код компилироваться не будет. Сначала необходимо включить заголовочный файл с необходимыми определениями для типов, встречающихся в `DriverEntry`. Один из возможных вариантов выглядит так:

```
#include <ntddk.h>
```

Шансы на успешную компиляцию кода повышаются, но попытка все равно завершится неудачей. Причина заключается в том, что по умолчанию компилятор интерпретирует предупреждения как ошибки, а функция не использует переданные ей аргументы. Отключать интерпретацию предупреждений как ошибок не рекомендуется, потому что некоторые предупреждения могут быть замаскированными ошибками. Чтобы избавиться от таких предупреждений, нужно полностью удалить имена аргументов (или закомментировать их) — это нормально для файлов C++. Существует и другое, более классическое решение — использование макроса `UNREFERENCED_PARAMETER`:

```
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);
```

```
    return STATUS_SUCCESS;
}
```

Этот макрос ссылается на переданный аргумент, просто записывая его значение; все претензии компилятора снимаются, так как аргумент был «использован».

Теперь проект компилируется нормально, но при компоновке происходит ошибка. Дело в том, что функция `DriverEntry` должна использовать схему компоновки С, которая не используется по умолчанию в C++. Ниже приведена итоговая версия успешной сборки драйвера, состоящего только из функции `DriverEntry`:

```
extern "C"
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    return STATUS_SUCCESS;
}
```

В какой-то момент драйвер может быть выгружен. В этот момент все, что было сделано в функции `DriverEntry`, должно быть отменено. Если этого не сделать, происходит утечка — ядро не перейдет в нормальное состояние до следующей перезагрузки. Драйверы могут содержать функцию выгрузки, которая автоматически вызывается перед выгрузкой драйвера из памяти. Указатель на нее должен был задан в поле `DriverUnload` объекта драйвера:

```
DriverObject->DriverUnload = SampleUnload;
```

Функция выгрузки получает объект драйвера (тот же, который передавался `DriverEntry`) и возвращает `void`. Так как наш пример драйвера ничего не делает в отношении выделения ресурсов в `DriverEntry`, в функции выгрузки ничего делать не придется, поэтому ее пока можно оставить пустой:

```
void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}
```

Полный исходный код драйвера на текущий момент:

```
#include <ntddk.h>

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}

extern "C"
NTSTATUS
```

```
DriverEntry(_In_ PDRIVER_OBJECT DriverObject,
            _In_ PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    return STATUS_SUCCESS;
}
```

Установка и загрузка драйвера

Итак, файл драйвера Sample.sys успешно откомпилирован; теперь установим его в системе, а затем загрузим его. Обычно установка и загрузка драйверов осуществляются на виртуальной машине, чтобы избежать риска фатальных ошибок на основной машине. Вы можете либо выбрать этот путь, либо пойти на небольшой риск с этим минималистским драйвером.

Для установки программного драйвера, как и для установки службы пользовательского режима, необходимо вызвать функцию API `CreateService` с правильными аргументами или же воспользоваться существующими инструментами. Один из самых известных инструментов такого рода — Sc.exe, встроенная Windows-программа для работы со службами. Мы воспользуемся ею для установки и последующей загрузки драйвера. Учтите, что установка и загрузка драйверов является привилегированной операцией, выполнение которой обычно разрешается только администраторам.

Откройте окно командной строки с повышенными привилегиями и введите следующую команду (в последней части следует указать фактический путь к файлу SYS в вашей системе):

```
sc create sample type= kernel binPath= c:\dev\sample\x64\debug\sample.sys
```

Обратите внимание: между словом `type` и знаком равенства нет пробела, а между знаком равенства и словом `kernel` пробел есть. То же относится и ко второй части.

Если все прошло нормально, в выходных данных должна быть выведена информация об успехе. Чтобы проверить установку, откройте редактор реестра (regedit.exe) и найдите драйвер в разделе `HKLM\System\CurrentControlSet\Services\Sample`.

На рис. 2.3 показан скриншот редактора реестра после выполнения приведенной выше команды.

Чтобы загрузить драйвер, снова воспользуйтесь программой Sc.exe, но на этот раз с параметром `Start`; параметр использует функцию API `StartService` для загрузки драйвера (эта же функция API используется для загрузки служб).

Тем не менее в 64-разрядных системах драйверы должны иметь цифровую подпись, поэтому обычно следующая команда завершится неудачей:

```
sc start sample
```

Так как подписывать драйвер в ходе разработки может быть неудобно (и даже невозможно, если у вас нет соответствующего сертификата), будет лучше перевести систему в режим тестовой подписи. В этом режиме неподписанные драйверы загружаются без проблем.

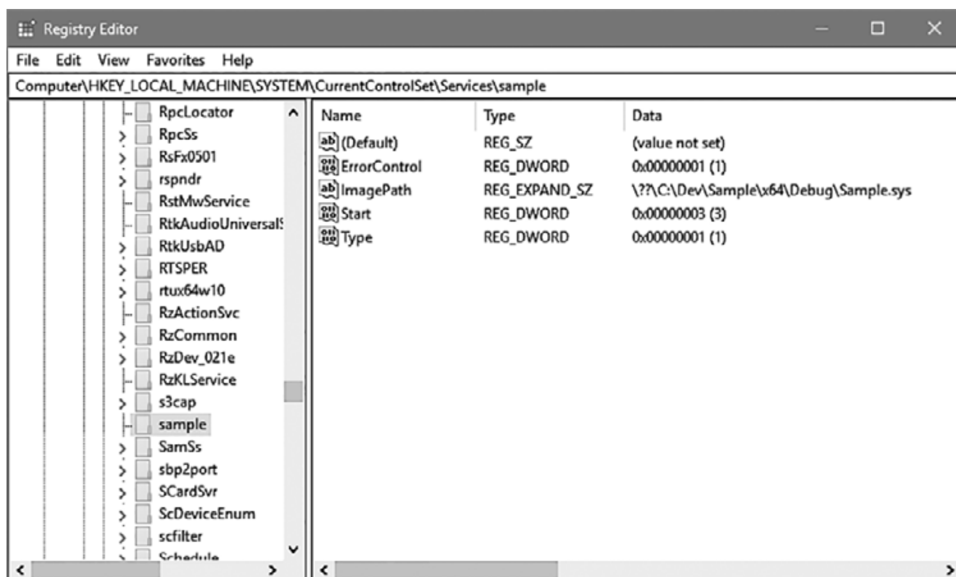


Рис. 2.3. Раздел реестра для установленного драйвера

В окне командной строки с повышенными привилегиями режим тестовой подписи включается командой следующего вида:

```
bcdedit /set testsigning on
```

К сожалению, команда вступает в силу после перезагрузки системы. После перезагрузки приведенная выше команда `start` должна работать успешно.

Если вы тестируете драйвер в Windows 10 с включенным режимом безопасной загрузки Secure Boot, попытка изменения режима тестовой подписи завершится неудачей. Это одна из настроек, защищенных режимом Secure Boot (также защищена локальная отладка режима ядра). Если вы не можете отключить режим Secure Boot в настройках BIOS из-за IT-политики в вашей организации или по другой причине, лучше всего проводить тестирование на виртуальной машине.

Есть еще одна настройка, которая может оказаться необходимой, если вы собираетесь тестировать драйвер в системе, предшествующей Windows 10. В этом случае следует задать целевую версию ОС в диалоговом окне свойств проекта (рис. 2.4). Обратите внимание: на иллюстрации я выбрал все конфигурации и все платформы, поэтому при переключении конфигурации (`Debug`/`Release`) или платформы (x86/x64/ARM/ARM64) настройка будет сохранена.

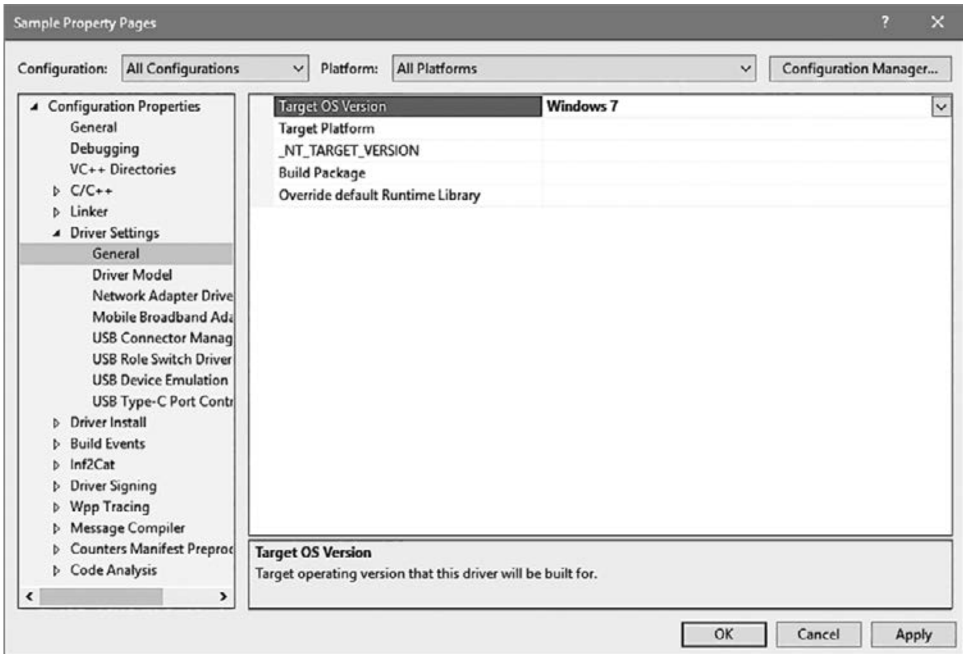


Рис. 2.4. Выбор целевой ОС в свойствах проекта

При включенном режиме тестовой подписи после загрузки драйвера должен быть выведен следующий результат:

```
SERVICE_NAME: sample
        TYPE : 1 KERNEL_DRIVER
        STATE : 4 RUNNING
                  (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE : 0 (0x0)
        SERVICE_EXIT_CODE : 0 (0x0)
        CHECKPOINT : 0x0
        WAIT_HINT : 0x0
        PID : 0
        FLAGS :
```

Это означает, что все хорошо, а драйвер загружен. Чтобы убедиться в этом, откройте Process Explorer и найдите файл образа драйвера Sample.sys. На рис. 2.5

выведена подробная информация об образе драйвера, загруженном в системное пространство.

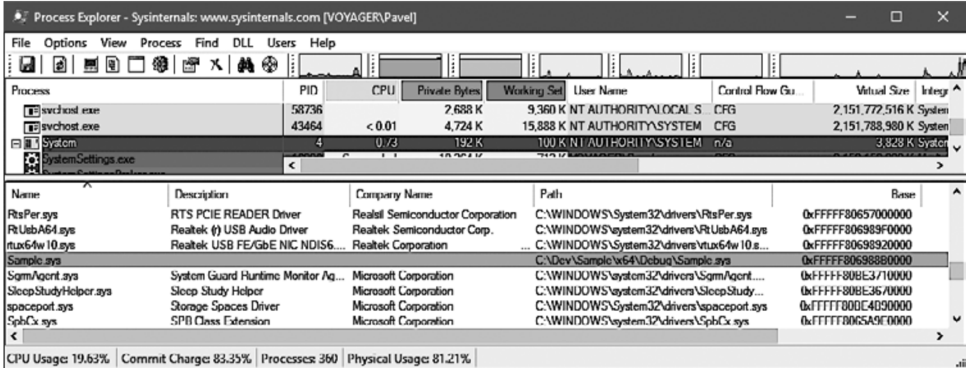


Рис. 2.5. Образ драйвера, загруженный в системное пространство

В этот момент драйвер можно выгрузить следующей командой:

```
sc stop sample
```

Во внутренней реализации `sc.exe` вызывает функцию API `ControlService` со значением `SERVICE_CONTROL_STOP`.

При выгрузке драйвера будет вызвана функция выгрузки, которая на данный момент не делает ничего. Чтобы убедиться в том, что драйвер был действительно выгружен, снова загляните в Process Explorer; на этот раз образ драйвера не должен присутствовать в списке.

Простая трассировка

Как убедиться в том, что функция `DriverEntry` и функция выгрузки были действительно выполнены? Добавим в эти функции простейшую трассировку. Драйверы могут использовать макрос `KdPrint` для вывода в стиле `printf` текста, который можно просматривать в отладчике ядра и других инструментах. Макрос `KdPrint` компилируется только в отладочных (`Debug`) сборках; он вызывает функцию API ядра `DbgPrint`.

Обновленные версии функции `DriverEntry` и функции выгрузки, использующие макрос `KdPrint` для трассировки факта выполнения их кода:

```
void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
```

```
    KdPrint(("Sample driver Unload called\n"));
}

extern "C"
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject,
            _In_ PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    KdPrint(("Sample driver initialized successfully\n"));

    return STATUS_SUCCESS;
}
```

Обратите внимание на двойные круглые скобки при использовании `KdPrint`. Они необходимы, потому что `KdPrint` является макросом, но при этом может получать произвольное количество аргументов в стиле `printf`. Так как макросы не могут получать переменное количество аргументов, для вызова функции `DbgPrint` используется трюк компилятора.

С этими командами можно загрузить драйвер снова и просмотреть эти сообщения. Отладчик ядра будет использоваться в главе 4, а пока мы воспользуемся удобной программой DebugView из пакета Sysinternals. Прежде чем запускать DebugView, необходимо провести подготовку. Прежде всего, начиная с Windows Vista, вывод DbgPrint генерируется только при наличии определенного значения в реестре. Вы должны добавить в реестр раздел с именем `Debug Print Filter` в разделе `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager` (обычно этот раздел не существует). Добавьте в новый раздел параметр `DWORD` с именем `DEFAULT` (не путайте со значением по умолчанию, существующим в любом разделе) и присвойте ему значение 8 (строго говоря, подойдет любое значение с установленным битом 3). На рис. 2.6 показан этот параметр в RegEdit. К сожалению, этот параметр вступит в силу только после перезагрузки системы.

После того как настройка вступит в силу, запустите DebugView (DbgView.exe) с повышенными привилегиями. В меню Options включите режим Capture Kernel (или нажмите Ctrl+K). Режимы Capture Win32 и Capture Global Win32 можно безопасно отключить, чтобы вывод разных процессов не загромождал экран.

Постройте драйвер, если это не было сделано ранее. Теперь можно загрузить драйвер в окне командной строки с повышенными привилегиями (`sc start sample`). Вывод в DebugView должен выглядеть так, как показано на рис. 2.7. При выгрузке драйвера появится другое сообщение из-за вызова функции выгрузки. (Третья строка вывода сгенерирована другим драйвером и не имеет отношения к нашему примеру.)

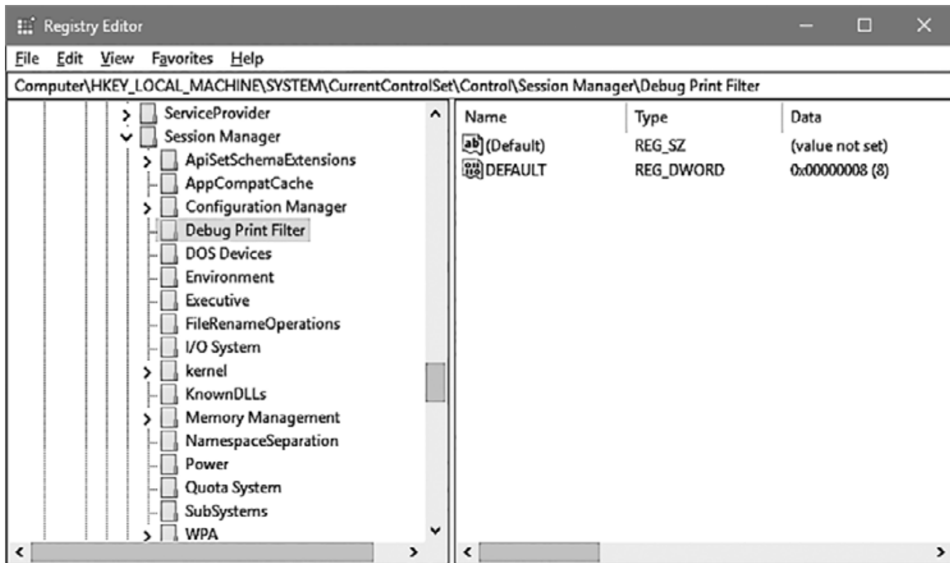


Рис. 2.6. Раздел Debug Print Filter в реестре

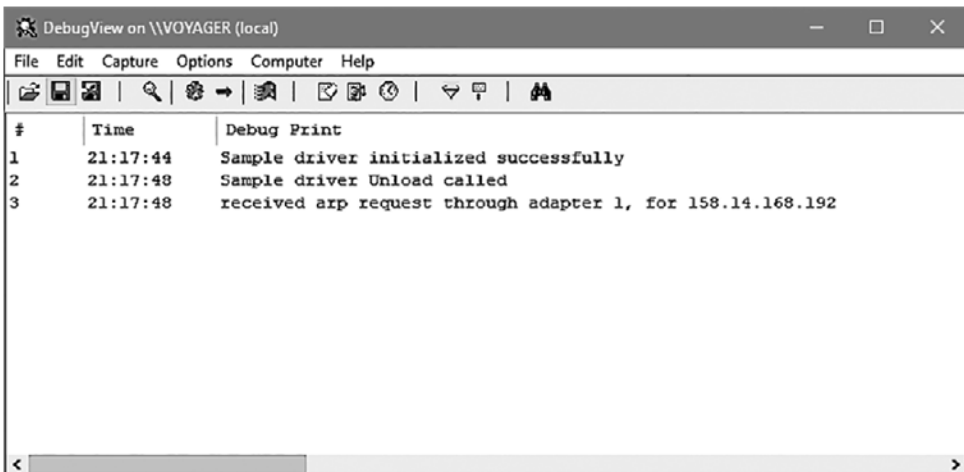


Рис. 2.7. Вывод программы DebugView из пакета Sysinternals

Упражнения

1. Добавьте в функцию `DriverEntry` код для вывода версии ОС Windows: основная версия, дополнительная версия и номер сборки. Используйте функцию `RtlGetVersion` для получения нужной информации. Проверьте результаты при помощи DebugView.

Итоги

В этой главе мы рассмотрели инструменты, необходимые для разработки ядра, а также написали минимальный драйвер, который доказывает работоспособность этих инструментов. В следующей главе будут рассмотрены фундаментальные структурные элементы функций API, концепций и структур режима ядра.

Глава 3

Основы программирования ядра

В этой главе более подробно рассматриваются функции API, структуры и определения режима ядра. Также будут описаны некоторые механизмы выполнения кода в драйвере. Наконец, вся новая информация будет объединена для построения нашего первого функционального драйвера.

В этой главе:

- Общие рекомендации программирования ядра
- Отладочные и конечные сборки
- API режима ядра
- Функции и коды ошибок
- Строки
- Динамическое выделение памяти
- Списки
- Объект драйвера
- Объекты устройств

Общие рекомендации программирования ядра

Для разработки драйверов ядра необходим пакет Windows Driver Kit (WDK) с соответствующими заголовочными файлами и библиотеками. API режима ядра состоит из функций C, которые имеют много общего с функциями пользовательского режима. Впрочем, есть и некоторые различия. В табл. 3.1 приве-

дена сводка основных различий между программированием пользовательского режима и программированием режима ядра.

Таблица 3.1. Различия в разработке для режима ядра и пользовательского режима

	Пользовательский режим	**Режим ядра**
Необработанное исключение	Фатальный сбой процесса	Фатальный сбой системы
Завершение	При завершении процесса вся приватная память и ресурсы освобождаются автоматически	Если драйвер выгружается без освобождения всех используемых им ресурсов, возникает утечка, которая будет исправлена только при следующей загрузке
Возвращаемые значения	Ошибки API иногда игнорируются	Ошибки (почти) никогда не должны игнорироваться
IRQL	Всегда `PASSIVE_LEVEL (0)`	Может быть `DISPATCH_LEVEL (2)` и выше
Ошибки программирования	Обычно локализуются в процессе	Могут иметь общесистемные последствия
Тестирование и отладка	Тестирование и отладка обычно выполняются на машине разработчика	Отладка должна выполняться на другой машине
Библиотеки	Может использовать практически любые библиотеки C/C++ (например, STL и boost)	Не может использовать большинство стандартных библиотек
Обработка исключений	Может использовать исключения C++ или структурированную обработку исключений (SEH)	Может использовать только SEH
Использование C++	Доступна вся среда времени выполнения C++	Среда времени выполнения C++ недоступна

Необработанные исключения

Исключения, происходящие в пользовательском режиме и не перехваченные программой, приводят к преждевременному завершению программы. С другой стороны, код режима ядра, который неявно считается пользующимся доверием, не может восстановиться из необработанного исключения. Такое исключение приведет к общему сбою системы с печально известным «синим экраном смерти», или BSOD (Blue Screen Of Death) (в новых версиях Windows

экраны общего сбоя используют более разнообразные цвета). На первый взгляд BSOD создает неудобства, но по сути это защитный механизм. Дело в том, что возможное продолжение выполнения кода может причинить непоправимый вред Windows (например, удаление важных файлов или повреждение реестра), из-за которого система не сможет загрузиться. А значит, лучше немедленно остановить работу, чтобы предотвратить возможные повреждения. BSOD более подробно рассматривается в главе 6.

Все сказанное ведет к одному простому выводу: код режима ядра следует писать очень внимательно и осторожно, уделяя особое внимание всем мелочам и проверкам ошибок.

Завершение

Когда процесс завершается по любой причине — нормальным образом, из-за необработанного исключения или из внешнего кода, — он никогда не оставляет после себя никаких утечек: вся приватная память освобождается, все дескрипторы закрываются и т. д. Конечно, преждевременное закрытие дескрипторов может привести к потере некоторых данных (например, при закрытии дескриптора файла перед записью части данных на диск), но утечки ресурсов не будет: это гарантируется ядром.

С другой стороны, драйверы ядра такой гарантии не дают. Если драйвер выгружается в тот момент, когда он все еще удерживает выделенную память или открытые дескрипторы режима ядра, такие ресурсы не будут освобождены автоматически. Освобождение произойдет только при следующей загрузке системы.

Почему это происходит? Разве ядро не может отслеживать выделение памяти и использование ресурсов драйверами, чтобы автоматически освобождать их при выгрузке драйвера?

Теоретически это возможно (хотя в настоящее время ядро не следит за использованием ресурсов). Настоящая проблема в том, что такие попытки освобождения ресурсов со стороны ядра были бы слишком опасными. Представьте, что драйвер выделил буфер в памяти, а затем передал его другому драйверу, с которым он взаимодействует. Второй драйвер использует буфер в памяти и в конечном итоге освобождает его. Если ядро попытается освободить буфер при выгрузке первого драйвера, то во втором драйвере попытка обращения к освобожденному буферу приведет к нарушению прав доступа и общему сбою системы.

Этот пример снова подчеркивает, что драйвер ядра должен тщательно прибрать за собой: никто другой этого не сделает.

Возвращаемые значения функций

В типичном коде пользовательского режима значения, возвращаемые функциями API, иногда игнорируются. Разработчик оптимистично полагает, что неудачная попытка вызова функции маловероятна. Насколько уместно такое предположение? Это зависит от конкретной функции. В худшем случае необработанное исключение приведет к сбою процесса, однако система не пострадает.

Игнорирование значений, возвращаемых функциями API ядра, намного более опасно (см. выше раздел «Завершение»), и обычно так поступать не рекомендуется. Даже «невинные» на первый взгляд функции могут завершаться неудачей по самым неожиданным причинам, поэтому золотое правило гласит: всегда проверяйте значения, возвращаемые функциями API ядра.

IRQL

Уровень запроса прерывания, или IRQL (Interrupt Request Level), важная концепция режима ядра, которая будет дополнительно рассматриваться в главе 6. А пока достаточно сказать, что обычно уровень IRQL процессора равен нулю, а точнее, он всегда равен нулю при выполнении кода пользовательского режима. В режиме ядра он остается равным нулю большую часть времени, но не всегда. Эффект ненулевых значений IRQL будет рассмотрен в главе 6.

Использование C++

В программировании пользовательского режима язык C++ использовался уже много лет, и он все еще хорошо работает в сочетании с вызовами функций API пользовательского режима. Компания Microsoft начала официально поддерживать C++ для кода режима ядра, начиная с Visual Studio 2012 и WDK 8. Конечно, язык C++ необязателен, но у него есть важные преимущества, связанные с освобождением ресурсов при использовании идиомы C++ RAII (Resource Acquisition Is Initialization, то есть «получение ресурса есть инициализация»). Мы будем неоднократно использовать идиому RAII для предотвращения утечки ресурсов.

Язык C++ почти в полной мере поддерживается для кода ядра. Однако для ядра не существует среды времени выполнения C++, поэтому некоторые возможности C++ просто не могут использоваться:

- Операторы `new` и `delete` не поддерживаются, а содержащий их код не компилируется. Это связано с тем, что обычно эти операторы выделяют память в куче пользовательского режима, что, конечно, неактуально для кода режима ядра. В функциях API режима ядра существуют функции-«заменители»,

построенные по образцу функций C `malloc` и `free`. Эти функции будут рассмотрены позднее в этой главе. Тем не менее эти операторы можно перегрузить по аналогии с тем, как это делается в C++ пользовательского режима, и вызывать в них функции выделения и освобождения функций режима ядра. Данная возможность также будет продемонстрирована позднее в этой главе.

- Глобальные переменные с конструкторами, отличными от конструкторов по умолчанию, вызываться не будут — нет никого, кто мог бы вызвать эти конструкторы. Таких ситуаций можно избежать несколькими способами:
  - Не включать код в конструктор; вместо этого создается функция инициализации `Init`, которая вызывается напрямую из кода драйвера (например, из `DriverEntry`).
  - Создать указатель в виде глобальной переменной и строить фактический экземпляр динамически. Компилятор будет генерировать правильный код вызова конструктора. Такое решение предполагает, что операторы `new` и `delete` были перегружены так, как описано позднее в этой главе.
- Ключевые слова обработки исключений C++ (`try`, `catch`, `throw`) не компилируются. Это объясняется тем, что механизму обработки исключений C++ нужна собственная среда времени выполнения, которая отсутствует в ядре. Обработка исключений может осуществляться только с использованием SEH (Structured Exception Handling) — специального механизма режима ядра. SEH подробно рассматривается в главе 6.
- Стандартные библиотеки C++ недоступны в режиме ядра. И хотя они в значительной степени основаны на шаблонах, код компилироваться не будет, потому что они зависят от библиотек и семантики пользовательского режима. При этом шаблоны C++ как языковая возможность работают нормально. Например, они могут использоваться для создания альтернативных типов для библиотечных типов пользовательского режима — `std::vector<>`, `std::wstring` и т. д.

Язык C++ используется в ряде примеров, приведенных в книге. Чаще всего используются следующие возможности:

- Ключевое слово `nullptr`, представляющее указатель `NULL`.
- Ключевое слово `auto` для автоматического определения типов при объявлении и инициализации переменных. Делает код более компактным, сокращает количество вводимых символов и позволяет сосредоточиться на важных аспектах кода.
- Шаблоны могут использоваться там, где это оправданно.
- Перегрузка операторов `new` и `delete`.
- Конструкторы и деструкторы, особенно для построения RAII-типов.

Строго говоря, драйверы можно без каких-либо проблем писать на «чистом» языке C. Если вы предпочитаете этот путь, используйте файлы с расширением C вместо CPP. Для таких файлов будет автоматически запускаться компилятор C.

Тестирование и отладка

В коде пользовательского режима тестирование обычно осуществляется на машине разработчика (если удовлетворены все необходимые зависимости). Отладка обычно выполняется присоединением отладчика (чаще всего Visual Studio) к выполняемому процессу (или процессам).

В коде режима ядра тестирование обычно осуществляется на другой машине — обычно на виртуальной машине, созданной на машине разработчика. Это гарантирует, что при возникновении критической ошибки BSOD машина разработчика не пострадает.

Отладка кода режима ядра должна выполняться на другой машине. Дело в том, что в режиме ядра при переходе к точке прерывания останавливается вся машина, а не конкретный процесс. Это означает, что на машине разработчика выполняется сам отладчик, тогда как на второй машине (обычно виртуальной) выполняется код драйвера. Две машины должны быть каким-то образом соединены, чтобы данные могли передаваться между хостом (на котором выполняется отладчик) и целевой машиной. Отладка режима ядра более подробно рассматривается в главе 5.

Отладочные и конечные сборки

Как и проекты пользовательского режима, драйверы режима ядра могут строиться в отладочном (`Debug`) или конечном (`Release`) режиме. Отличия между сборками так же, как между аналогами пользовательского режима — отладочная сборка по умолчанию не использует оптимизации, но она проще отлаживается. В конечных сборках используются оптимизации компилятора для генерирования самого быстрого кода. Впрочем, это еще не все.

В терминологии режима ядра используются термины «проверяемая» (Checked) и «свободная» (Free). Хотя в проектах режима ядра Visual Studio продолжают использоваться термины «отладочная/конечная», в старой документации используются термины «проверяемая/свободная». С точки зрения компиляции отладочная сборка режима ядра определяет символическую переменную `DBG` и присваивает ему значение 1 (по аналогии с символической переменной `_DEBUG`, определяемой в пользовательском режиме). Это означает, что по символической переменной `DBG` можно отличать отладочные сборки от конечных с использо-

ванием средств условной компиляции. Собственно, именно это делает макрос `KdPrint`: в отладочной версии он компилируется в вызов `DbgPrint`, а в конечной версии он компилируется в ничто, вследствие чего вызовы `KdPrint` ничего не делают в конечных сборках.

Обычно это именно то, что вам требуется, потому что такие вызовы обходятся относительно дорого. Другие способы сохранения информации в журнале описаны в главе 10.

API режима ядра

Драйверы режима ядра используют функции, экспортируемые компонентами режима ядра. Такие функции образуют *API режима ядра*. Большинство функций API реализуется непосредственно в ядре (NtOskrnl.exe), но некоторые могут быть реализованы в других модулях ядра, например в HAL (hal.dll).

API режима ядра представляет собой большой набор функций C. Имена многих функций начинаются с префикса — признака компонента, реализующего функцию. В табл. 3.2 приведены самые распространенные префиксы и их смысл.

Таблица 3.2. Распространенные префиксы функций API режима ядра

Префикс	Смысл	Пример
`Ex`	Общие функции исполнительной среды	`ExAllocatePool`
`Ke`	Общие функции режима ядра	`KeAcquireSpinLock`
`Mm`	Диспетчер памяти	`MmProbeAndLockPages`
`Rtl`	Общая библиотека времени выполнения	`RtlInitUnicodeString`
`FsRtl`	Библиотека времени выполнения файловой системы	`FsRtlGetFileSize`
`Flt`	Библиотека мини-фильтров файловой системы	`FltCreateFile`
`Ob`	Диспетчер объектов	`ObReferenceObject`
`Io`	Диспетчер ввода/вывода	`IoCompleteRequest`
`Se`	Безопасность	`SeAccessCheck`
`Ps`	Структура процессов	`PsLookupProcessByProcessId`
`Po`	Диспетчер электропитания	`PoSetSystemState`
`Wmi`	Инструментарий управления Windows	`WmiTraceMessage`
`Zw`	Платформенные обертки API	`ZwCreateFile`
`Hal`	Уровень абстрагирования оборудования	`HalExamineMBR`
`Cm`	Диспетчер конфигурации (реестр)	`CmRegisterCallbackEx`

Просматривая список экспортируемых функций NtOsKrnl.exe, вы найдете в нем функции, не документированные в Windows Driver Kit; это просто факт из жизни разработчика режима ядра — не вся информация документирована.

Сейчас заслуживает особого упоминания один набор — функции с префиксами `Zw`. Такие функции повторяют платформенные функции API, реализованные в NtDll.dll как шлюзы, тогда как их фактическая реализация находится в исполнительной среде. При вызове функции с префиксом `Nt` (например, `NtCreateFile`) из пользовательского режима функция обращается за фактической реализацией `NtCreateFile` к исполнительной среде. На этой стадии `NtCreateFile` может выполнить различные проверки на основании того факта, что исходный вызов поступил из пользовательского режима. Информация о вызывающей стороне хранится на уровне отдельных потоков в недокументированном поле `PreviousMode` в структуре `KTHREAD` для каждого потока.

С другой стороны, если драйверу режима ядра потребуется вызвать системную сервисную функцию, на такой вызов не должны распространяться проверки и ограничения, применяемые для вызовов из пользовательского режима. Здесь в игру вступают функции `Zw`. Вызов функции `Zw` устанавливает для предыдущей вызывающей стороны режим `KernelMode` (0), после чего вызывает платформенную функцию. Например, вызов `ZwCreateFile` назначает предыдущей вызывающей стороне значение `kernelMode`, после чего вызывает `NtCreateFile`, в результате чего `NtCreateFile` обходит часть проверок безопасности и буферов, которые были бы выполнены в обычном случае. Таким образом, драйверы режима ядра должны вызывать функции `Zw`, если только нет веских причин для обратного.

Функции и коды ошибок

Большинство функций API режима ядра возвращают признак успеха или неудачи для операции. Признак имеет тип `NTSTATUS` — 32-разрядное целое со знаком. Значение `STATUS_SUCCESS` (0) указывает на удачное выполнение операций. Отрицательные значения являются признаками ошибок. Все определенные значения `NTSTATUS` можно найти в файле ntstatus.h. Для многих ветвей кода точная природа ошибки несущественна, поэтому достаточно проверить только старший бит. Это можно сделать при помощи макроса `NT_SUCCESS`. Следующий пример проверяет ошибку, и если она будет обнаружена, сохраняет информацию в журнале:

```
NTSTATUS DoWork() {
    NTSTATUS status = CallSomeKernelFunction();
    if(!NT_SUCCESS(Statue)) {
        KdPirnt((L"Error occurred: 0x%08X\n", status));
        return status;
```

```
    }

    // Другие операции

    return STATUS_SUCCESS;
}
```

Иногда значения NTSTATUS, возвращаемые функциями, в конечном итоге возвращаются в пользовательский режим. В таких случаях значение STATUS_xxx преобразуется в значение ERROR_yyy, которое может быть получено в пользовательском режиме вызовом функции GetLastError. Учтите, что их числовые значения не совпадают: во-первых, код ошибки в пользовательском режиме имеет положительные значения, во-вторых, соответствие не является однозначным. Как бы то ни было, для драйвера ядра это обычно не создает проблем.

Внутренние функции драйвера ядра тоже обычно возвращают код NTSTATUS для обозначения своего статуса успеха/неудачи. Обычно это удобно, потому что эти функции вызывают функции API режима ядра и могут распространить любую ошибку, просто вернув код статуса, полученный от конкретной функции API. Также эта схема подразумевает, что «настоящие» возвращаемые значения функций драйверов обычно возвращаются через указатели и ссылки, передаваемые в аргументах функции.

Строки

В API режима ядра часто используются строки. В некоторых случаях эти строки представляют собой простые указатели на символы Юникода (wchar_t* или одно из определений типов — например, WCHAR), но большинство функций, работающих со строками, работают со структурой типа UNICODE_STRING.

> Термин «Юникод» в этой книге считается приблизительно эквивалентным UTF-16: один символ кодируется 2 байтами. Этот способ хранения используется во внутреннем представлении компонентов ядра.

Структура UNICODE_STRING представляет строку с известной длиной и максимальной длиной. Упрощенное определение этой структуры выглядит так:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;
```

Значение поля длины Length задается в байтах (не в символах). В него не включается NULL-завершитель Юникода, если он существует (NULL-завершитель не является строго обязательным). Поле MaximumLength содержит количество байтов, до которого может вырасти строка без необходимости повторного выделения памяти.

Для манипуляций со структурами UNICODE_STRING обычно используются функции Rtl, предназначенные для работы со строками. Некоторые часто используемые функции этой категории перечислены в табл. 3.3.

Таблица 3.3. Основные функции UNICODE_STRING

Функция	**Описание**
RtlInitUnicodeString	Инициализирует структуру UNICODE_STRING на основании указателя на существующую строку C. Сначала функция заполняет Buffer, затем вычисляет Length и присваивает MaximumLength то же значение. Обратите внимание: эта функция не выделяет память — она только инициализирует внутренние поля
RtlCopyUnicodeString	Копирует одну структуру UNICODE_STRING в другую. Память указателя на приемную строку (Buffer) должны быть выделена до копирования, а значение MaximumLength задано соответствующим образом
RtlCompareUnicodeString	Сравнивает две структуры UNICODE_STRING (меньше, равно, больше) с указанием того, должен ли при сравнении учитываться регистр символов
RtlEqualUnicodeString	Проверяет две структуры UNICODE_STRING на равенство с указанием того, должен ли при сравнении учитываться регистр символов
RtlAppendUnicodeString ToString	Присоединяет одну структуру UNICODE_STRING к другой
RtlAppendUnicodeToString	Присоединяет UNICODE_STRING к строке в стиле C

Кроме перечисленных функций, также существуют функции, которые работают с указателями на строки C. Более того, некоторые хорошо известные функции из библиотеки времени выполнения C для удобства также реализованы в ядре: wcscpy, wcscat, wcslen, wcscpy_s, wcschr, strcpy, strcpy_s и другие.

Префикс wcs работает со строками C в Юникоде, а префикс str работает со строками C в ANSI. Суффикс _s у некоторых функций обозначает безопасную функцию; таким функциям передается дополнительный аргумент с максимальной длиной строки, чтобы функция не пыталась передавать данные, выходящие за границу буфера.

Динамическое выделение памяти

Драйверам часто требуется выделять память динамически. Как обсуждалось в главе 1, размер стека ядра достаточно невелик, поэтому большие блоки памяти должны выделяться динамически.

Ядро предоставляет два обобщенных пула памяти для использования драйверами (эти пулы также используются ядром):

- *Выгружаемый* (paged) пул — пул памяти, который может выгружаться из памяти при необходимости.
- *Невыгружаемый* (non-paged) пул — пул памяти, который никогда не выгружается и гарантированно остается в ОЗУ.

Очевидно, невыгружаемый пул «лучше» выгружаемого, поскольку он никогда не может стать причиной ошибки отсутствия страницы в памяти. Позднее в книге будет показано, что некоторые ситуации требуют выделения памяти из невыгружаемого пула. Драйверы должны использовать этот пул по минимуму, только когда это неизбежно. Во всех остальных ситуациях драйверы должны использовать выгружаемый пул. Перечисление `POOL_TYPE` содержит многочисленные «типы» пулов, но только три из них должны использоваться драйверами: `PagedPool`, `NonPagedPool` и `NonPagedPoolNx` (невыгружаемый пул без разрешений на исполнение).

В табл. 3.4 приведена сводка самых полезных функций, используемых при работе с пулами памяти ядра.

Таблица 3.4. Функции для работы с пулами памяти ядра

Функция	**Описание**
`ExAllocatePool`	Выделяет память в одном из пулов с тегом по умолчанию. Эта функция считается устаревшей, вместо нее следует использовать следующую функцию в таблице
`ExAllocatePoolWithTag`	Выделяет память в одном из пулов с заданным тегом
`ExAllocatePoolWithQuotaTag`	Выделяет память в одном из пулов с заданным тегом и начисляет выделенную память в квоту текущего процесса
`ExFreePool`	Освобождает выделенную память. Пул, из которого была выделена память, определяется функцией автоматически

Некоторые функции получают аргумент, который позволяет пометить выделенную память 4-байтовым тегом. Обычно значение содержит до 4 ASCII-символов, логически описывающих драйвер или его отдельную часть. Теги

могут использоваться для выявления утечек памяти — если память, помеченная тегом драйвера, остается и после выгрузки драйвера. Для просмотра выделенных блоков памяти (и их тегов) можно воспользоваться программой Poolmon WDL или моей программой PoolMonX (загружается по адресу http://www.github.com/zodiacon/AllTools). На рис. 3.1 изображен снимок экрана PoolMonX.

Рис. 3.1. PoolMonX

Следующий пример демонстрирует возможности выделения памяти и копирования строк для сохранения пути реестра, переданного `DriverEntry`, и освобождения этой строки в функции выгрузки:

```
// Определяет тег (из-за обратного порядка байтов
// при просмотре в PoolMon отображается в виде ‘abcd’)
#define DRIVER_TAG 'dcba'
UNICODE_STRING g_RegistryPath;
extern "C" NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject,
            _In_ PUNICODE_STRING RegistryPath) {
    DriverObject->DriverUnload = SampleUnload;

    g_RegistryPath.Buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool,
        RegistryPath->Length, DRIVER_TAG);
    if (g_RegistryPath.Buffer == nullptr) {
        KdPrint(("Failed to allocate memory\n"));
```

```
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    g_RegistryPath.MaximumLength = RegistryPath->Length;
    RtlCopyUnicodeString(&g_RegistryPath, (PCUNICODE_STRING)RegistryPath);

    // %wZ для объектов UNICODE_STRING
    KdPrint(("Copied registry path: %wZ\n", &g_RegistryPath));
    //...
    return STATUS_SUCCESS;
}

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    ExFreePool(g_RegistryPath.Buffer);
    KdPrint(("Sample driver Unload called\n"));
}
```

Списки

Режим ядра использует циклические двусвязные списки во многих внутренних структурах данных. Например, всеми процессами в системе управляют структуры `EPROCESS`, объединенные в циклический двусвязный список, заголовок которого хранится в переменной ядра `PsActiveProcessHead`.

Все эти списки строятся сходным образом. Центральное место в них занимает структура `LIST_ENTRY`, определяемая следующим образом:

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

На рис. 3.2 изображен пример такого списка с заголовком и тремя экземплярами.

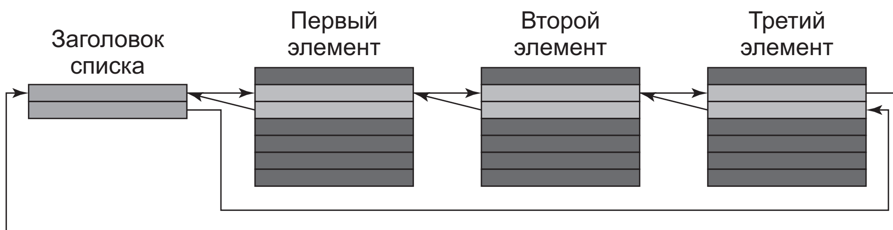


Рис. 3.2. Циклический двусвязный список

Одна такая структура встраивается в реальную структуру, которая представляет для вас интерес. Например, в структуре EPROCESS поле ActiveProcessLinks относится к типу LIST_ENTRY и содержит указатели на следующий и предыдущий объекты LIST_ENTRY других структур EPROCESS. Заголовок списка хранится отдельно; для процессов это PsActiveProcessHead. Чтобы получить указатель на фактическую структуру, представляющий интерес, по известному адресу LIST_ENTRY, можно воспользоваться макросом CONTAINING_RECORD.

Допустим, вы хотите вести список структур типа MyDataItem, которые определяются следующим образом:

```
struct MyDataItem {
    // Некоторые поля данных
    LIST_ENTRY Link;
    // Другие поля данных
};
```

При работе с такими связными списками у вас имеется заголовок списка, хранящийся в переменной. Это означает, что естественный обход элементов списка осуществляется по полю Flink списка, содержащему указатель на следующую структуру LIST_ENTRY в списке. Однако при получении указателя на LIST_ENTRY нас в действительности интересует структура MyDataItem, в одном из полей которой хранится эта структура. Для получения этой структуры используется макрос CONTAINING_RECORD:

```
MyDataItem* GetItem(LIST_ENTRY* pEntry) {
    return CONTAINING_RECORD(pEntry, MyDataItem, Link);
}
```

Макрос вычисляет правильное смещение и выполняет преобразование к фактическому типу данных (MyDataItem в этом примере).

В табл. 3.5 перечислены основные функции для работы со связными списками. Все операции выполняются с постоянным временем.

Таблица 3.5. Функции для работы с циклическими связными списками

Функция	Описание
InitializeListHead	Инициализирует заголовок списка для создания пустого списка. Указатели на следующий и предыдущий элемент инициализируются указателем на следующий элемент
InsertHeadList	Вставляет элемент в начало списка
InsertTailList	Вставляет элемент в конец списка
IsListEmpty	Проверяет, является ли список пустым
RemoveHeadList	Удаляет элемент в начале списка

Функция	Описание
RemoveTailList	Удаляет элемент в конце списка
RemoveEntryList	Удаляет конкретный элемент из списка
ExInterlockedInsertHeadList	Выполняет атомарную вставку элемента в начало списка с использованием заданной спин-блокировки
ExInterlockedInsertTailList	Выполняет атомарную вставку элемента в конец списка с использованием заданной спин-блокировки
ExInterlockedRemoveHeadList	Выполняет атомарное удаление элемента в начале списка с использованием заданной спин-блокировки

Последние три функции в табл. 3.4 выполняют атомарные операции с использованием примитива синхронизации, называемого спин-блокировкой. Спин-блокировки будут рассматриваться в главе 6.

Объект драйвера

Вы уже знаете, что функция `DriverEntry` получает два аргумента; в первом передается объект драйвера. Он представляет собой полудокументированную структуру `DRIVER_OBJECT`, определенную в заголовках WDK. «Полудокументированность» означает, что не все поля структуры документированы для использования драйверами. Структура создается ядром и частично инициализируется, а затем передается `DriverEntry` (а также функции выгрузки перед выгрузкой драйвера). В этой точке драйвер должен провести дополнительную инициализацию структуры, чтобы показать, какие операции поддерживаются драйвером.

Одна такая «операция» уже упоминалась в главе 2 — это функция выгрузки. Другой важный набор инициализируемых операций составляют функции диспетчеризации — это массив указателей на функции, хранящийся в поле `MajorFunction` структуры `DRIVER_OBJECT`. Этот набор функций указывает, какие конкретные операции поддерживает драйвер: создание, чтение, запись и т. д. Коды функций определяются значениями с префиксом `IRP_MJ_`. В табл. 3.6 приведены коды первичных функций и их смысл.

Изначально ядро инициализирует массив `MajorFunction` указателями на внутреннюю функцию ядра `IopInvalidDeviceRequest`, которая возвращает вызывающей стороне статус ошибки — признак того, что операция не поддерживается. Это означает, что драйвер в своей функции `DriverEntry` должен инициализировать только реально поддерживаемые операции, а всем остальным элементам следует оставить значения по умолчанию.

Таблица 3.6. Коды первичных функций ядра

Первичная функция	Описание
IRP_MJ_CREATE (0)	Операция создания. Обычно используется для вызовов CreateFile или ZwCreateFile
IRP_MJ_CLOSE (2)	Операция закрытия. Обычно используется для вызовов CloseHandle или ZwCloseHandle
IRP_MJ_READ (3)	Операция чтения. Обычно используется для ReadFile, ZwReadFile и аналогичных функций чтения
IRP_MJ_WRITE (4)	Операция записи. Обычно используется для WriteFile, ZwWriteFile и аналогичных функций чтения
IRP_MJ_DEVICE_CONTROL (14)	Обобщенный вызов драйвера, используется для вызовов DeviceIoControl или ZwDeviceIoControlFile
IRP_MJ_INTERNAL_DEVICE_ CONTROL (15)	Аналог предыдущей операции, но только для вызова из режима ядра
IRP_MJ_PNP (31)	Обратный вызов Plug and Play, используемый диспетчером Plug and Play. Обычно представляет интерес для драйверов оборудования или фильтров таких драйверов
IRP_MJ_POWER (22)	Обратный вызов управления питанием, используемый диспетчером электропитания. Обычно представляет интерес для драйверов оборудования или фильтров таких драйверов

Например, наш драйвер Sample пока не поддерживает никаких функций диспетчеризации; это означает, что с ним невозможно взаимодействовать. Драйвер должен поддерживать как минимум операции IRP_MJ_CREATE и IRP_MJ_CLOSE, чтобы иметь возможность открыть один из объектов устройств. Эти концепции будут применены на практике в следующей главе.

Объекты устройств

Может показаться, что объект драйвера — хороший кандидат для взаимодействия с клиентами, но это не так. Конечными точками, через которые клиент общается с драйверами, являются *объекты устройств* — экземпляры полудокументированной структуры DEVICE_OBJECT. Без объектов устройств такое взаимодействие невозможно. А следовательно, драйвер должен создать хотя бы один объект устройства и присвоить ему имя, чтобы клиенты могли общаться с ним.

Функция CreateFile (и ее разновидности) получает первый аргумент, который называется «именем файла», но в действительности должен содержать указатель на имя объекта устройства (а файл всего лишь является частным случаем).

Вообще говоря, имя функции `CreateFile` выбрано неудачно — под «файлом» в данном случае имеется в виду объект файла. При открытии дескриптора для файла или устройства создается экземпляр `FILE_OBJECT` — еще одной полудокументированной структуры ядра.

А если говорить еще точнее, `CreateFile` получает символическую ссылку (symbolic link) — объект ядра, который может указывать на другой объект ядра. (Концептуально символическая ссылка напоминает ярлыки (shortcuts) файловой системы.) Все символические ссылки, которые могут использоваться в вызовах `CreateFile` и `CreateFile2` пользовательского режима, находятся в каталоге диспетчера объектов с именем `??`. Для его просмотра можно воспользоваться программой WinObj из пакета Sysinternals. На рис. 3.3 показан этот каталог (в WinObj он отображается под именем `Global??`).

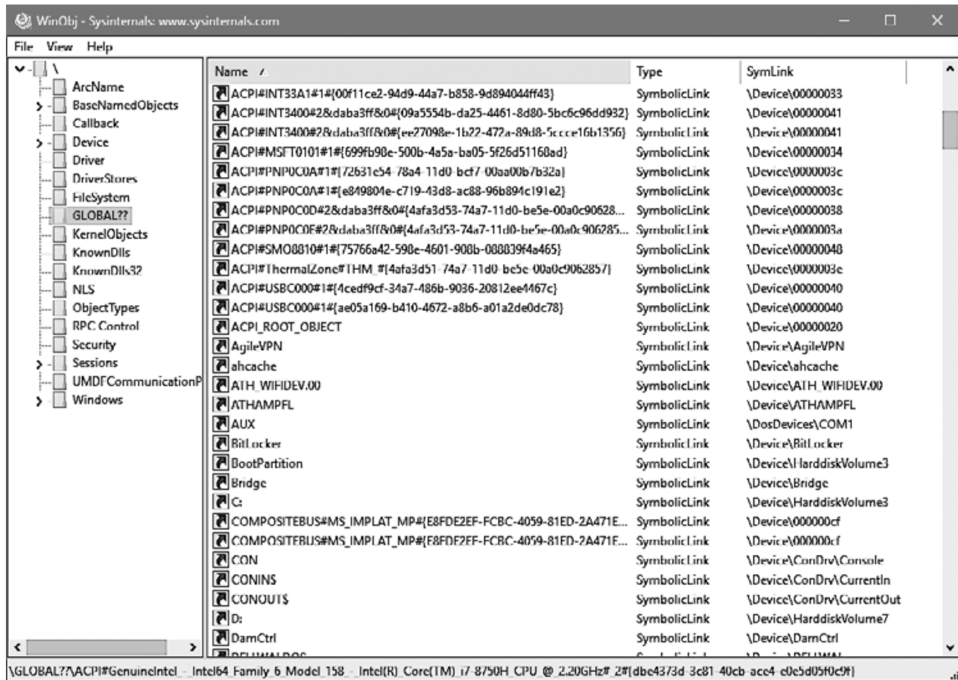


Рис. 3.3. Каталог символических ссылок в WinObj

Некоторые имена выглядят знакомо: `C:`, `Aux`, `Con` и т. д. Для вызовов `CreateFile` они являются действительными «именами файлов». Другие элементы — длинные загадочные строки — генерируются системой ввода/вывода для драйверов оборудования, вызывающих функцию API `IoRegisterDeviceInterface`. Эти типы символических ссылок не относятся к теме книги.

Большинство символических ссылок в каталоге `??` указывает на внутреннее имя устройства из каталога `Device`. Имена в этом каталоге недоступны напрямую для вызовов из пользовательского режима. Впрочем, к ним можно обращаться из режима ядра при помощи функции API `IoGetDeviceObjectPointer`.

Каноническим примером служит драйвер программы Process Explorer. При запуске с административными правами Process Explorer устанавливает драйвер, который предоставляет Process Explorer возможности, выходящие за рамки функций API пользовательского режима (даже при запуске с повышенными привилегиями). Например, Process Explorer в своем окне Thread для процесса может вывести полный стек вызовов потока, включая функции в режиме ядра. Получить такую информацию из пользовательского режима не удастся; недостающую информацию предоставляет драйвер.

Драйвер, устанавливаемый Process Explorer, создает один объект ядра, чтобы программа Process Explorer могла открыть дескриптор для этого устройства и выдавать запросы. Это означает, что объект устройства должен быть именованным, а в каталоге `??` для него должна присутствовать символическая ссылка; она действительно присутствует здесь под именем `PROCEXP152` — вероятно, для драйвера версии 15.2 (на момент написания этой книги). На рис. 3.4 изображена эта символическая ссылка в WinObj.

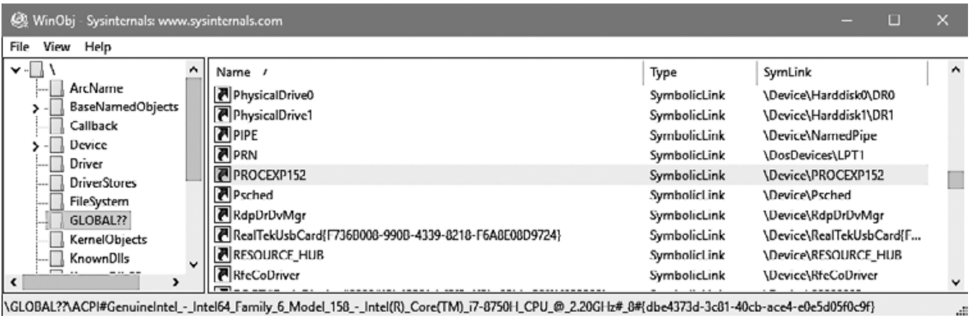


Рис. 3.4. Символическая ссылка Process Explorer в WinObj

Обратите внимание: символическая ссылка устройства Process Explorer указывает на `\Device\PROCEXP152` — внутреннее имя, доступное только для вызовов из режима ядра. Вызов `CreateFile`, выполняемый из Process Explorer (или любого другого клиента) по символической ссылке, должен быть снабжен префиксом `\\.\`. Это необходимо для того, чтобы парсер диспетчера объектов не решил, что строка «PROCEXP152» относится к файлу в текущем каталоге. Вот как Process Explorer открывает дескриптор для своего устройства (обратите внимание на удвоение символа `\` — это необходимо из-за использования обратного слеша в качестве экранирующего символа):

```
HANDLE hDevice = CreateFile(L"\\\\.\\PROCEXP152",
    GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);
```

Драйвер создает объект устройства функцией `IoCreateDevice`. Эта функция выделяет память и инициализирует структуру объекта устройства и возвращает указатель на него на сторону вызова. Экземпляр объекта устройства хранится в поле `DeviceObject` структуры `DRIVER_OBJECT`. Если создается более одного объекта устройства, они образуют односвязный список, в котором поле `NextDevice` структуры `DEVICE_OBJECT` указывает на следующий объект устройства. Обратите внимание: объекты устройств вставляются в начало списка, так что первый созданный объект устройства хранится в последней позиции; его поле `NextDevice` содержит ссылку `NULL`. Эти связи изображены на рис. 3.5.

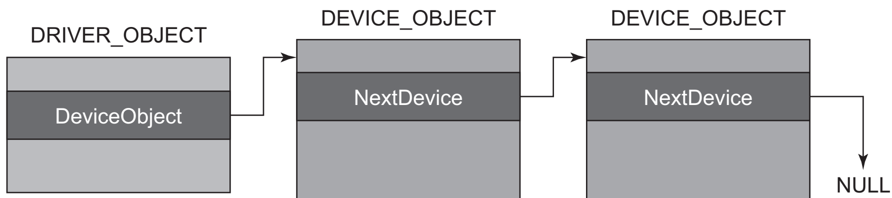


Рис. 3.5. Объекты драйвера и устройства

Итоги

В этой главе были рассмотрены некоторые фундаментальные структуры данных и функции API ядра. В следующей главе мы построим более функциональный драйвер и клиента, а также рассмотрим материал этой главы на более глубоком уровне.

Глава 4
Драйвер: от начала до конца

В этой главе многие концепции, представленные ранее, используются для построения простого, но полнофункционального драйвера и клиентского приложения. При этом будут дополнены некоторые подробности, пропущенные ранее. Мы установим драйвер и воспользуемся его функциональностью для выполнения операции режима ядра, недоступной для пользовательского режима.

В этой главе:

- Введение
- Инициализация драйвера
- Клиентский код
- Функции диспетчеризации `Create` и `Close`
- Функция диспетчеризации `DeviceIoControl`
- Установка и тестирование

Введение

Проблема, которую мы будем решать при помощи простого драйвера режима ядра, — негибкая система назначения приоритетов потоков средствами Windows API. В пользовательском режиме приоритет потока определяется комбинацией класса приоритета его процесса со смещением на уровне отдельного потока, которое имеет ограниченное количество уровней.

Изменение класса приоритета процесса может осуществляться функцией `SetPriorityClass`, которая получает дескриптор процесса и один из шести поддерживаемых классов приоритетов. Каждый класс приоритета соответствует уровню приоритета, который определяет приоритет по умолчанию

для всех потоков, создаваемых в этом процессе. Приоритет конкретного потока может быть изменен функцией SetThreadPriority, которая получает дескриптор потока и одну из нескольких констант, соответствующих смещениям относительно базового класса приоритета. В табл. 4.1 показаны доступные приоритеты потоков для класса приоритета процесса и смещения приоритета потока.

Таблица 4.1. Допустимые значения приоритетов потоков для Windows API

Класс приоритета	**–Sat**	**–2**	**–1**	**0 (по умолчанию)**	**+1**	**+2**	**+Sat**	**Комментарии**
Низкий приоритет	1	2	3	4	5	6	15	
Приоритет ниже нормального	1	4	5	6	7	8	15	
Нормальный приоритет	1	6	7	8	9	10	15	
Приоритет выше нормального	1	8	9	10	11	12	15	
Высокий приоритет	1	11	12	13	14	15	15	Доступны только шесть уровней (не семь)
Приоритет реального времени	16	22	23	24	25	26	31	Могут быть выбраны все уровни от 16 до 31

Значения, передаваемые SetThreadPriority, задают смещение. Пять уровней соответствуют смещениям от –2 до +2: THREAD_PRIORITY_LOWEST (–2), THREAD_PRIORITY_BELOW_NORMAL (–1), THREAD_PRIORITY_NORMAL (0), THREAD_PRIORITY_ABOVE_NORMAL (+1), THREAD_PRIORITY_HIGHEST (+2). Два оставшихся уровня, называемые *уровнями насыщения*, задают приоритету два крайних значения, поддерживаемых для данного класса приоритета: THREAD_PRIORITY_IDLE (-Sat) и THREAD_PRIORITY_TIME_CRITICAL (+Sat).

Следующий пример кода меняет приоритет текущего потока на 11:

```
SetPriorityClass(GetCurrentProcess(), ABOVE_NORMAL_PRIORITY_CLASS);
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_ABOVE_NORMAL);
```

Таблица 4.1 довольно наглядно демонстрирует проблему, которую мы хотим решить. Лишь небольшой набор приоритетов может назначаться напрямую. Хотелось бы создать драйвер, который позволит обойти эти ограничения и назначить приоритету потока любое числовое значение независимо от его класса.

Существование класса приоритета реального времени не означает, что Windows является ОС реального времени; Windows не предоставляет некоторых временных гарантий, которые обычно предоставляются настоящими операционными системами реального времени. Кроме того, поскольку приоритеты реального времени очень высоки и обычно конкурируют со многими потоками ядра, выполняющими важную работу, такие процессы должны выполняться с привилегиями администратора; в противном случае при попытке назначения класса приоритета реального времени будет назначен высокий приоритет.

Между приоритетами реального времени и более низкими классами приоритетов существуют и другие различия. За дополнительной информацией обращайтесь к книге «Внутреннее устройство Windows». 7-е изд.

Инициализация драйвера

Начнем с построения драйвера точно так же, как это делалось в главе 2. Создайте новый проект типа WDM Empty Project с именем `PriorityBooster` (или выберите другое имя на свое усмотрение) и удалите INF-файл, созданный мастером. Затем добавьте в проект новый исходный файл с именем PriorityBooster.cpp (или другим именем, которое вы выбрали). Добавьте базовую директиву `#include` для основного заголовка WDK и пустую функцию `DriverEntry`:

```
#include <ntddk.h>

extern "C" NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject,
            _In_ PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

Многие драйверы должны выполнять в `DriverEntry` следующие действия:

- Назначить функцию выгрузки.
- Задать функции диспетчеризации, поддерживаемые драйвером.
- Создать объект устройства.
- Создать символическую ссылку на объект устройства.

После того как эти операции будут выполнены, драйвер готов к получению запросов.

Начнем с добавления функции выгрузки и сохранения указателя на нее в объекте драйвера. Новая версия `DriverEntry` с функцией выгрузки:

```
// Прототипы

void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject);
```

```
// DriverEntry

extern "C" NTSTATUS

DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath)
{
    DriverObject->DriverUnload = PriorityBoosterUnload;

    return STATUS_SUCCESS;
}

void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
}
```

Код в функцию выгрузки будет добавляться по мере надобности, когда в `DriverEntry` будет выполняться реальная работа, последствия которой нужно будет отменить.

А теперь необходимо задать функции диспетчеризации, которые мы собираемся поддерживать. Практически все драйверы должны поддерживать `IRP_MJ_CREATE` и `IRP_MJ_CLOSE`, в противном случае драйвер не сможет открыть дескриптор какого-либо устройства для этого драйвера. В `DriverEntry` добавляется следующий код:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = PriorityBoosterCreateClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = PriorityBoosterCreateClose;
```

Первичным функциям `Create` и `Close` присваиваются указатели на одну функцию. Это объясняется тем, что, как вскоре будет показано, они, по сути, будут делать одно и то же: просто одобрять запрос. В более сложных случаях это могут быть разные функции, а в случае `Create` драйвер может (например) проверить, от кого поступил вызов, и разрешить открытие устройства только проверенным вызывающим сторонам.

Все первичные функции имеют одинаковый прототип (они являются частью массива указателей на функции), поэтому в программу нужно добавить прототип для `PriorityBoosterCreateClose`. Прототип для этих функций выглядит так:

```
NTSTATUS PriorityBoosterCreateClose(_In_ PDEVICE_OBJECT DeviceObject,
                                    _In_ PIRP Irp);
```

Функция должна вернуть `NTSTATUS` и получать указатель на объект устройства и указатель на пакет запроса ввода/вывода IRP (I/O Request Packet). IRP — основной объект для хранения информации о запросе любого типа. Объекты IRP будут более подробно рассмотрены в главе 6, но основы будут рассмотрены позднее в этой главе, так как они необходимы для завершения работы над драйвером.

Передача информации драйверу

Настроенные выше операции `Create` и `Close` необходимы, но, конечно, их недостаточно. Нужно каким-то образом сообщить драйверу, какому потоку и какое значение приоритета следует присвоить. С точки зрения клиента пользовательского режима существуют три базовые функции, которые он может использовать: `WriteFile`, `ReadFile` и `DeviceIoControl`.

Для целей нашего драйвера можно использовать либо `WriteFile`, либо `DeviceIoControl`. Операция чтения смысла не имеет, потому что мы передаем информацию драйверу, а не получаем ее от драйвера. Что же лучше, `WriteFile` или `DeviceIoControl`? В основном это дело вкуса, но обычно рекомендуется использовать `WriteFile` для операций, которые фактически являются операциями записи (на логическом уровне); для всех остальных целей предпочтительна функция `DeviceIoControl`, так как она предоставляет общий механизм передачи данных драйверу и от него.

Так как изменение приоритета потока не является чистой операцией записи, мы выберем `DeviceIoControl`. Прототип этой функции выглядит так:

```
BOOL WINAPI DeviceIoControl(
    _In_ HANDLE hDevice,
    _In_ DWORD dwIoControlCode,
    _In_reads_bytes_opt_(nInBufferSize) LPVOID lpInBuffer,
    _In_ DWORD nInBufferSize,
    _Out_writes_bytes_to_opt_(nOutBufferSize,*lpBytesReturned)
                              LPVOID lpOutBuffer,
    _In_ DWORD nOutBufferSize,
    _Out_opt_ LPDWORD lpBytesReturned,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

`DeviceIoControl` получает три исключительно важных параметра:

- Код управляющей операции.
- Входной буфер.
- Выходной буфер.

Это означает, что функция `DeviceIoControl` предоставляет гибкий механизм взаимодействия с драйверами. Возможна поддержка разных кодов управляющих операций, что потребует разной семантики с передачей дополнительных буферов.

На стороне драйвера `DeviceIoControl` соответствует коду первичной функции `IRP_MJ_DEVICE_CONTROL`. Добавим ее в инициализацию функций диспетчеризации:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = PriorityBoosterDeviceControl;
```

Протокол обмена данными между клиентом и драйвером

Раз мы решили использовать `DeviceIoControl` для взаимодействия между клиентом и драйвером, теперь необходимо определить фактическую семантику. Очевидно, потребуется код управляющей операции и входной буфер. Этот буфер должен содержать значения, необходимые драйверу для выполнения операции: идентификатор потока и приоритет, который этому потоку назначается.

Оба значения должны использоваться как на стороне драйвера, так и на стороне клиента. Клиент должен предоставить данные, а драйвер — работать с ними. Это означает, что определения должны храниться в отдельном файле, который включается как кодом драйвера, так и клиентским кодом.

Для этой цели мы добавим в проект драйвера заголовочный файл с именем PriorityBoosterCommon.h. Этот файл также будет использоваться позднее в клиенте пользовательского режима.

В файл необходимо включить два определения: определение структуры данных, которую драйвер ожидает получить от клиентов, и код управляющей операции для изменения приоритета потока. Начнем с объявления структуры для информации, необходимой драйверу:

```
struct ThreadData {
    ULONG ThreadId;
    int Priority;
};
```

В полях структуры хранится уникальный идентификатор потока и целевой приоритет. Идентификаторы потоков представляют собой 32-разрядные целые без знака, поэтому мы выбираем тип `ULONG` (учтите, что нормально использовать `DWORD` — стандартный тип, определенный в заголовках пользовательского режима, — не удастся, потому что он не определен в заголовках режима ядра; с другой стороны, тип `ULONG` определяется в обоих режимах). Приоритет должен быть числом от 1 до 31, так что простого 32-разрядного целого будет достаточно.

Теперь необходимо определить код управляющей операции. Казалось бы, подойдет любое 32-разрядное число, но это не так. Код управляющей операции должен строиться макросом `CTL_CODE`, который получает четыре аргумента, образующие итоговый код. Макрос `CTL_CODE` определяется следующим образом:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

Краткое описание аргументов макроса:

- `DeviceType` — определяет тип устройства. Это может быть одна из констант `FILE_DEVICE_xxx`, определяемых в заголовках WDK, но в основном они

предназначены для драйверов оборудования. Для программных драйверов (таких, как наш) конкретное число особой роли не играет. Тем не менее в документации Microsoft сказано, что значения для сторонних драйверов должны начинаться с 0x8000.

- `Function` — возрастающее число, обозначающее конкретную операцию. Как минимум, это число должно быть разным для разных кодов управляющих операций одного драйвера. И снова подойдет любое число, но в официальной документации сказано, что сторонние драйверы должны начинаться с 0x800.
- `Method` — самая важная часть кода управляющей операции. Она указывает, как входной и выходной буферы, предоставленные клиентом, передаются драйверу. Эти значения будут более подробно разобраны в главе 6. Для нашего драйвера будет использоваться простейшее значение `METHOD_NEITHER`. Его эффект будет рассмотрен позднее в этой главе.
- `Access` — указывает, обращена ли операция к драйверу (`FILE_WRITE_ACCESS`), от драйвера (`FILE_READ_ACCESS`) или в обе стороны (`FILE_ANY_ACCESS`). Типичный драйвер просто использует `FILE_ANY_ACCESS` и разбирается с фактическим запросом в обработчике `IRP_MJ_DEVICE_CONTROL`.

На основании этой информации единственный код управляющей операции можно определить следующим образом:

```
#define PRIORITY_BOOSTER_DEVICE 0x8000

#define IOCTL_PRIORITY_BOOSTER_SET_PRIORITY CTL_CODE(PRIORITY_BOOSTER_DEVICE, \
    0x800, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Создание объекта устройства

Впрочем, инициализации в `DriverEntry` еще не закончены. В настоящее время объекта устройства еще нет, поэтому мы не сможем открыть дескриптор и обратиться к драйверу. Типичному программному драйверу достаточно всего одного объекта устройства, на которое указывает символическая ссылка, чтобы клиенты пользовательского режима могли получать дескрипторы.

Для создания объекта устройства необходимо вызвать функцию API `IoCreateDevice`, объявление которой выглядит так (некоторые аннотации SAL опущены/упрощены для ясности):

```
NTSTATUS IoCreateDevice(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ ULONG DeviceExtensionSize,
    _In_opt_ PUNICODE_STRING DeviceName,
    _In_ DEVICE_TYPE DeviceType,
    _In_ ULONG DeviceCharacteristics,
```

```
    _In_ BOOLEAN Exclusive,
    _Outptr_ PDEVICE_OBJECT *DeviceObject);
```

Аргументы IoCreateDevice описаны ниже:

- DriverObject — объект драйвера, которому принадлежит объект устройства. Это должен быть просто объект драйвера, переданный функции DriverEntry.
- DeviceExtensionSize — дополнительные байты, которые будут выделены к sizeof(DEVICE_OBJECT). Данная возможность полезна для того, чтобы связать с устройством некоторую структуру данных. Для программных драйверов, создающих всего один объект устройства, она менее актуальна, потому что состоянием, необходимым для устройства, можно управлять из глобальных переменных.
- DeviceName — внутреннее имя устройства, обычно создаваемое в каталоге Device диспетчера объектов.
- DeviceType — актуально для некоторых разновидностей драйверов оборудования. Для программных драйверов должно использоваться значение FILE_DEVICE_UNKNOWN.
- DeviceCharacteristics — набор флагов, актуальных для некоторых конкретных драйверов. Программные драйверы должны использовать 0 или FILE_DEVICE_SECURE_OPEN, если они поддерживают истинное пространство имен (которые редко используются программными драйверами — впрочем, эта тема выходит за рамки материала книги).
- Exclusive — разрешено ли открывать более одного объекта файла для открытия одного и того же устройства? Для большинства драйверов должно использоваться значение FALSE, но в некоторых ситуациях значение TRUE будет более уместно; оно ограничивает каждое устройство только одним клиентом.
- DeviceObject — возвращенный указатель, передаваемый в форме указателя на указатель. При успешном выполнении IoCreateDevice выделяет структуру из невыгружаемого пула памяти и сохраняет полученный указатель внутри разыменованного аргумента.

Перед вызовом IoCreateDevice необходимо создать структуру UNICODE_STRING для хранения внутреннего имени устройства:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\PriorityBooster");
// RtlInitUnicodeString(&devName, L»\\Device\\ThreadBoost»);
```

Имя устройства может быть любым, но оно должно находиться в каталоге Device диспетчера объектов. Есть два способа инициализации UNICODE_STRING строковой константой. В первом используется функция RtlInitUnicodeString,

которая работает вполне нормально. Однако функция RtlInitUnicodeString должна подсчитать количество символов в строке, чтобы правильно инициализировать поля Length и MaximumLength. В данном случае это не создаст проблем, но существует более быстрый способ — использование макроса RTL_CONSTANT_STRING, который вычисляет длину строки статически во время компиляции, а это означает, что он будет правильно работать только со строковыми константами.

Теперь можно вызвать функцию IoCreateDevice:

```
PDEVICE_OBJECT DeviceObject;
NTSTATUS status = IoCreateDevice(
    DriverObject // наш объект драйвера,
    0 // дополнительные байты не нужны,
    &devName // имя устройства,
    FILE_DEVICE_UNKNOWN // тип устройства,
    0 // флаги характеристик,
    FALSE // не монопольное использование,
    &DeviceObject // полученный указатель
    );
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create device object (0x%08X)\n", status));
    return status;
}
```

Если все прошло хорошо, теперь у нас появился указатель на объект устройства. На следующем шаге следует открыть доступ к объекту устройства вызывающим сторонам пользовательского режима, предоставив им символическую ссылку. Следующий фрагмент создает символическую ссылку и связывает ее с объектом устройства:

```
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
status = IoCreateSymbolicLink(&symLink, &devName);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
    IoDeleteDevice(DeviceObject);
    return status;
}
```

Для выполнения своей работы IoCreateSymbolicLink получает символическую ссылку и цель ссылки. Обратите внимание: если попытка создания завершится неудачей, необходимо отменить все, что было сделано ранее (в данном случае только создание объекта устройства), вызовом IoDeleteDevice. В более общем смысле, если DriverEntry вернет какой-либо признак неудачи, функция выгрузки не должна вызываться. Если бы была выполнена более значительная работа по инициализации, было бы необходимо помнить об отмене всего сделанного до этого момента в случае неудачи. Более элегантное решение будет продемонстрировано в главе 5.

После того как символическая ссылка будет создана, а объект устройства подготовлен, `DriverEntry` может вернуть признак успеха. Драйвер готов принимать запросы.

Прежде чем двигаться дальше, стоит вспомнить о функции выгрузки. Если выполнение `DriverEntry` завершилось успешно, функция выгрузки должна отменить все, что было сделано в `DriverEntry`. В нашем случае это две операции: создание объекта устройства и создание символической ссылки. Операции должны отменяться в обратном порядке:

```
void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
    // Удалить символическую ссылку
    IoDeleteSymbolicLink(&symLink);

    // Удалить объект устройства
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

Клиентский код

К этому моменту стоит написать клиентский код пользовательского режима. Все необходимое для клиента уже было определено ранее.

Добавьте в решение новый консольный проект с именем Booster (или любым другим именем на ваше усмотрение). Мастер Visual Studio создает один исходный файл (Visual Studio 2019) или два предварительно откомпилированных заголовочных файла (pch.h, pch.cpp) в Visual Studio 2017. На предварительно откомпилированные заголовочные файлы пока можно не обращать внимания.

В файле Booster.cpp удалите код по умолчанию «hello, world» и добавьте следующее объявление:

```
#include <windows.h>
#include <stdio.h>
#include "..\PriorityBooster\PriorityBoosterCommon.h"
```

Обратите внимание на включение общего заголовочного файла, созданного драйвером и используемого совместно с клиентским кодом.

Измените функцию `main`, чтобы она получала аргументы командной строки. Приложение передает в аргументах идентификатор потока и приоритет и дает команду драйверу заменить приоритет потока заданным значением.

```
int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: Booster <threadid> <priority>\n");
        return 0;
    }
```

Затем необходимо открыть дескриптор для нашего устройства. «Именем файла» для CreateFile должна быть символическая ссылка с добавленным префиксом \\.\. Весь вызов должен выглядеть так:

```
HANDLE hDevice = CreateFile(L"\\\\.\\PriorityBooster", GENERIC_WRITE,
    FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE)
    return Error("Failed to open device");
```

Функция Error просто выводит некоторый текст с последней обнаруженной ошибкой:

```
int Error(const char* message) {
    printf("%s (error=%d)\n", message, GetLastError());
    return 1;
}
```

Вызов CreateFile должен обратиться к драйверу в своей функции диспетчеризации IRP_MJ_CREATE. Если драйвер не загружен в этот момент, то есть отсутствует объект устройства и символическая ссылка, будет получена ошибка с номером 2 (файл не найден). Теперь, когда у нас имеется действительный дескриптор устройства, можно перейти к подготовке вызова DeviceIoControl. Сначала нужно создать структуру ThreadData и заполнить подробности:

```
ThreadData data;
data.ThreadId = atoi(argv[1]); // Первый аргумент командной строки
data.Priority = atoi(argv[2]); // Второй аргумент командной строки
```

Все готово для вызова DeviceIoControl и последующего закрытия дескриптора устройства:

```
DWORD returned;
BOOL success = DeviceIoControl(hDevice,
    IOCTL_PRIORITY_BOOSTER_SET_PRIORITY, // Код управляющей операции
    &data, sizeof(data), // Входной буфер и длина
    nullptr, 0, // Выходной буфер и длина
    &returned, nullptr);
if (success)
    printf("Priority change succeeded!\n");
else
    Error("Priority change failed!");

CloseHandle(hDevice);
```

DeviceIoControl обращается к драйверу, вызывая первичную функцию IRP_MJ_DEVICE_CONTROL.

На данный момент клиентский код завершен. Остается только реализовать функции диспетчеризации, объявленные на стороне драйвера.

Функции диспетчеризации Create и Close

Теперь все готово к реализации трех функций диспетчеризации, определенных драйвером. Простейшими из них, безусловно, являются процедуры создания и закрытия. Все, что нужно, — это завершить запрос с успешным статусом. Вот полная реализация процедуры отправки/закрытия:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    UNREFERENCED_PARAMETER(DeviceObject);

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

Каждая функция диспетчеризации получает целевой объект устройства и пакет запроса ввода/вывода (IRP). Объект устройства нас не особо интересует, потому что он только один, и должен быть именно тот объект, который был создан в `DriverEntry`. С другой стороны, структура IRP исключительно важна. Структуры IRP более подробно рассматриваются в главе 6, но сейчас необходимо сказать несколько слов.

IRP — полудокументированная структура, представляющая запрос, который обычно исходит от одного из диспетчеров в исполнительной среде: диспетчера ввода/вывода, диспетчера Plug & Play или диспетчера электропитания. С простым программным драйвером это с большой вероятностью будет диспетчер ввода/вывода. Независимо от того, кто создал IRP, целью драйвера является обработка IRP, что подразумевает анализ подробностей запроса и выполнение действий, необходимых для его завершения.

Каждый запрос к драйверу всегда прибывает упакованным в пакет IRP, будь то `Create`, `Close`, `Read` или любой другой вид IRP. Проверяя поля IRP, можно определить тип и подробности запроса (обычно указатель на функцию диспетчеризации базируется на типе запроса, так что в большинстве случаев тип запроса уже известен). Стоит отметить, что пакет IRP никогда не приходит в одиночку; он сопровождается одной или несколькими структурами `IO_STACK_LOCATION`. В простых случаях (как в нашем драйвере) используется только одна структура `IO_STACK_LOCATION`. В сложных случаях выше или ниже нашего драйвера могут находиться драйверы-фильтры и существуют несколько экземпляров `IO_STACK_LOCATION`, по одному для каждого уровня в стеке устройства. (Эта возможность более подробно рассматривается в главе 6.) Проще говоря, часть необходимой информации находится в базовой структуре IRP, а другая часть — в структуре `IO_STACK_LOCATION` для нашего «уровня» в стеке устройств.

В случае `Create` и `Close` проверять значения полей не нужно. Достаточно задать статус IRP в поле `IoStatus` (типа `IO_STATUS_BLOCK`), которое также состоит из двух полей:

- `Status` — статус, с которым завершится этот запрос.
- `Information` — полиморфное поле, смысл которого изменяется в зависимости от запроса. В случае `Create` и `Close` достаточно нулевого значения.

Чтобы фактически завершить IRP, мы вызываем `IoCompleteRequest`. Эта функция должна сделать много всего, но по сути, она распространяет IRP обратно к создателю (обычно диспетчеру ввода/вывода), а диспетчер оповещает клиента о том, что операция завершена. Второй аргумент содержит временный прирост приоритета, который драйвер может обеспечить своему клиенту. В большинстве случаев нулевое значение работает лучше всего (`IO_NO_INCREMENT` определяется как 0), потому что запрос завершился синхронно, и нет никаких причин для наращивания приоритета вызывающей стороны. Дополнительная информация об этой функции также предоставляется в главе 6.

Остается выполнить последнюю операцию — вернуть тот же статус, который был помещен в IRP. На первый взгляд это кажется бесполезным дублированием, но это необходимо (причина будет объяснена в одной из следующих глав).

Функция диспетчеризации DeviceIoControl

Мы подошли к самой сути происходящего. Весь код драйвера, написанный до настоящего момента, приводил к этой функции диспетчеризации. Именно здесь выполняется вся реальная работа по назначению запрашиваемого приоритета заданному потоку.

Первое, что необходимо сделать, — проверить код управляющей операции. Типичные драйверы могут поддерживать много кодов управляющих операций, поэтому запрос должен немедленно завершиться неудачей, если код управляющей операции не опознан:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    // Получить IO_STACK_LOCATION
    auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
    auto status = STATUS_SUCCESS;

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
        case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
            // Выполнить основную работу
            break;
```

```
        default:
            status = STATUS_INVALID_DEVICE_REQUEST;
            break;
    }
```

Чтобы получить информацию о любом пакете IRP, необходимо заглянуть в структуру IO_STACK_LOCATION, связанную с текущим уровнем устройства. Вызов IoGetCurrentIrpStackLocation возвращает указатель на правильную структуру IO_STACK_LOCATION. В нашем случае существует всего одна структура IO_STACK_LOCATION, но в любом случае следует вызывать IoGetCurrentIrpStackLocation.

Главным компонентом IO_STACK_LOCATION является огромное поле-объединение Parameters, которое содержит набор структур — по одной для каждого типа IRP. В случае IRP_MJ_DEVICE_CONTROL представляет интерес структура DeviceIoControl. В этой структуре находится информация, переданная клиентом, например код управляющей операции, буферы и их длины.

Команда switch использует поле IoControlCode для определения того, распознан код управляющей операции или нет. Если код не распознан, мы просто устанавливаем статус, отличный от успеха, и выходим из блока switch.

Последний блок обобщенного кода, который нам понадобится, — завершение IRP после блока switch независимо от того, успешно он завершился или нет. В противном случае клиент не получит ответа о завершении:

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

IRP просто завершается с тем статусом, который окажется подходящим. Если код управляющей операции не был опознан, это будет статус неудачи. В противном случае он зависит от реальной работы, выполненной тогда, когда код управляющей операции был опознан.

Самым интересным и важным является последний фрагмент — выполнение реальной работы по изменению приоритета потока. Сначала нужно проверить, хватит ли размера полученного буфера для хранения объекта ThreadData. Указатель на входной буфер, предоставленный пользователем, хранится в поле Type3InputBuffer, а длина входного буфера — в поле InputBufferLength:

```
if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(ThreadData)) {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
}
```

Будем считать, что буфер имеет достаточный размер, поэтому его можно интерпретировать как ThreadData:

```
auto data = (ThreadData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
```

> Возможно, вас интересует, допустимо ли обращаться к переданному буферу? Так как буфер находится в пространстве пользовательского режима, выполнение должно происходить в контексте процесса клиента. И это условие выполняется, так как вызывающей стороной является сам поток клиента, перешедший в режим ядра так, как описано в главе 1.

Если указатель равен NULL, выполнение следует прервать:

```
if (data == nullptr) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
```

Затем проверим, лежит ли приоритет в допустимом диапазоне от 1 до 31, и если приоритет выходит за пределы диапазона, выполнение также будет прервано:

```
if (data->Priority < 1 || data->Priority > 31) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
```

Мы подходим ближе к своей цели. Функция API, которую нам хотелось бы использовать, называется KeSetPriorityThread. Ее прототип выглядит так:

```
KPRIORITY KeSetPriorityThread(
    _Inout_ PKTHREAD Thread,
    _In_ KPRIORITY Priority);
```

Тип KPRIORITY представляет собой 8-разрядное целое число. Сам поток идентифицируется указателем на объект KTHREAD — один из компонентов управления потоками в режиме ядра. Он полностью не документирован, но здесь важно то, что мы получили идентификатор потока от клиента и должны каким-то образом получить указатель на реальный объект потока в пространстве ядра. Функция, которая может найти поток по идентификатору, называется PsLookupThreadByThreadId. Чтобы получить ее определение, необходимо добавить еще одну директиву #include:

```
#include <ntifs.h>
```

Обратите внимание: эта директива #include должна быть добавлена до <ntddk.h>, в противном случае при компиляции произойдет ошибка.

Теперь можно преобразовать идентификатор потока в указатель:

```
PETHREAD Thread;
status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &Thread);
if (!NT_SUCCESS(status))
    break;
```

В этом фрагменте стоит обратить внимание на ряд важных моментов:

- Функция поиска получает `HANDLE` вместо идентификатора. Что же это — дескриптор или идентификатор? Это идентификатор, оформленный с типом дескриптора. Такое решение объясняется особенностями обработки и генерирования идентификаторов потоков. Идентификаторы генерируются по глобальной приватной таблице дескрипторов ядра, так что «значения» дескрипторов в действительности являются идентификаторами. Макрос `ULongToHandle` обеспечивает необходимое преобразование, чтобы предотвратить ошибки компиляции. (Напомним, что в 64-разрядных системах значение `HANDLE` является 64-разрядным, тогда как идентификатор потока, предоставленный клиентом, всегда является 32-разрядным.)
- Полученный указатель имеет тип `PETHREAD`, то есть «указатель на `ETHREAD`». Структура `ETHREAD` также является полностью недокументированной. Но похоже, тут возникает проблема, потому что `KeSetPriorityThread` получает `PKTHREAD` вместо `PETHREAD`. Оказывается, это одно и то же, потому что первым полем `ETHREAD` является `KTHREAD` (поле с именем `Tcb`). Все это будет доказано в следующей главе, когда мы будем использовать отладчик ядра. Из всего сказанного следует, что `PKTHREAD` при необходимости можно спокойно использовать вместо `PETHREAD` (и наоборот) без малейших проблем.
- Вызов `PsLookupThreadByThreadId` может завершиться неудачей по разным причинам, например из-за недействительного идентификатора потока или из-за того, что поток уже завершился. Если вызов завершился неудачей, программа просто выходит из `switch` со статусом, возвращенным из функции.

Наконец-то все готово к изменению приоритета. Но подождите: что, если после успешного последнего вызова поток завершится непосредственно перед назначением нового приоритета? Не беспокойтесь, такого быть не может. С технической точки зрения поток может завершиться в этот момент, но это не приведет к появлению висячего указателя. Дело в том, что функция поиска в случае успешного выполнения увеличивает счетчик ссылок для объекта потока режима ядра, поэтому он не может быть уничтожен до того, как мы явно уменьшим счетчик ссылок. Вызов изменения приоритета выглядит так:

```
KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
```

Остается уменьшить счетчик ссылок объекта потока; если этого не сделать, возникнет утечка ресурсов, которая будет исправлена только при следующей загрузке системы. Для решения этой задачи используется функция `ObDereferenceObject`:

```
ObDereferenceObject(Thread);
```

Все готово! Для удобства приведу полный код обработчика `IRP_MJ_DEVICE_CONTROL` с небольшими косметическими изменениями:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    // Получить IO_STACK_LOCATION
    auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
    auto status = STATUS_SUCCESS;

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY: {
        // Выполнить основную работу
        auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
        if (len < sizeof(ThreadData)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }

        auto data = (ThreadData*)stack->Parameters.DeviceIoControl.
Type3InputBuffer;
        if (data == nullptr) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }

        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }

        PETHREAD Thread;
        status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId),
&Thread);
        if (!NT_SUCCESS(status))
            break;

        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);
        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
        break;
    }

    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }

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

Установка и тестирование

На данный момент мы можем успешно построить драйвер и клиент. Следующим шагом должна стать установка драйвера и тестирование его функциональности. Следующее тестирование можно провести на виртуальной машине или, если вам хватит храбрости, — на машине разработки.

Начнем с установки драйвера. Откройте окно командной строки с повышенными привилегиями и установите драйвер при помощи программы sc.exe, как это было сделано в главе 2:

```
sc create booster type= kernel binPath= c:\Test\PriorityBooster.sys
```

Проследите за тем, чтобы значение `binPath` включало полный путь полученного SYS-файла. Имя драйвера (`booster`) в нашем примере является именем созданного раздела реестра, поэтому оно должно быть уникальным. Оно не обязано соответствовать имени SYS-файла.

Теперь драйвер можно загрузить:

```
sc start booster
```

Если все прошло нормально, драйвер должен запуститься успешно. Чтобы убедиться в этом, откройте WinObj и найдите имя устройства и символическую ссылку. На рис. 4.1 показана символическая ссылка в WinObj.

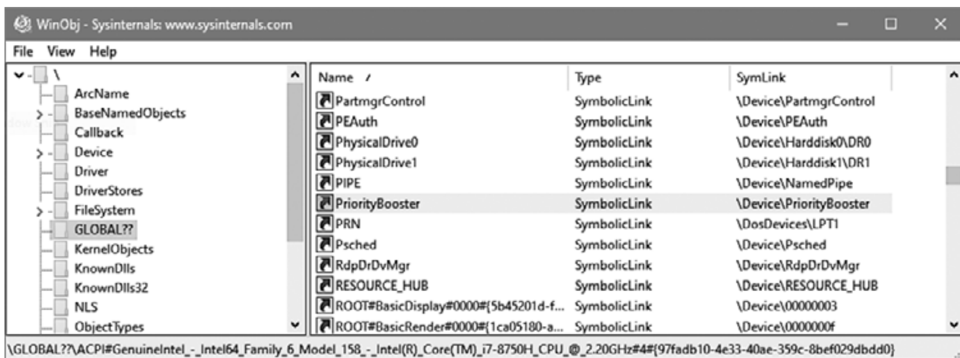


Рис. 4.1. Символическая ссылка в WinObj

А теперь запустим исполняемый файл клиента. На рис. 4.2 в окне Process Explorer показан поток процесса cmd.exe, выбранный в качестве примера, — мы изменим значение его приоритета.

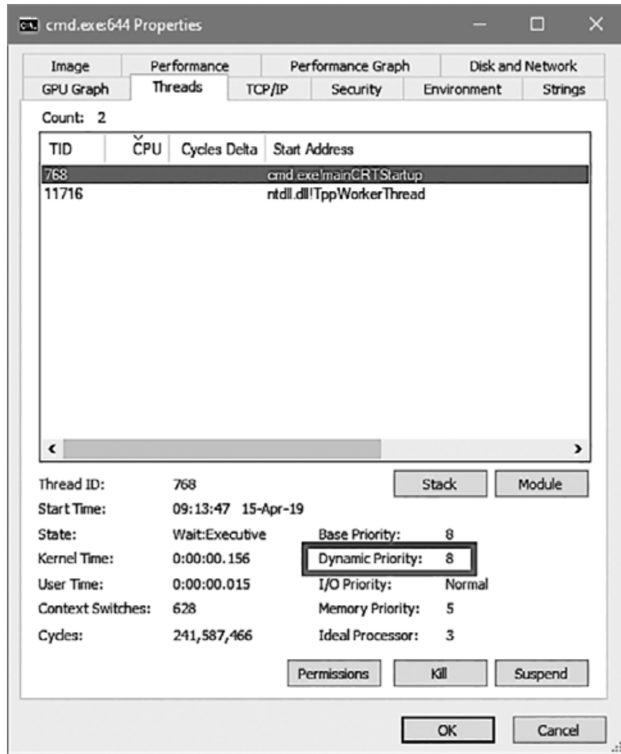


Рис. 4.2. Исходный приоритет потока

Запустите клиент с указанием идентификатора потока и нужного приоритета (замените идентификатор потока значением для вашей системы):

```
booster 768 25
```

Если при запуске произойдет ошибка, возможно, следует выбрать статическую библиотеку времени выполнения вместо DLL-библиотеки. Выберите свойства проекта, откройте узел C++ и в группе Code Generation выберите вариант `Multithreaded Debug`.

Вуаля! Результат показан на рис. 4.3.

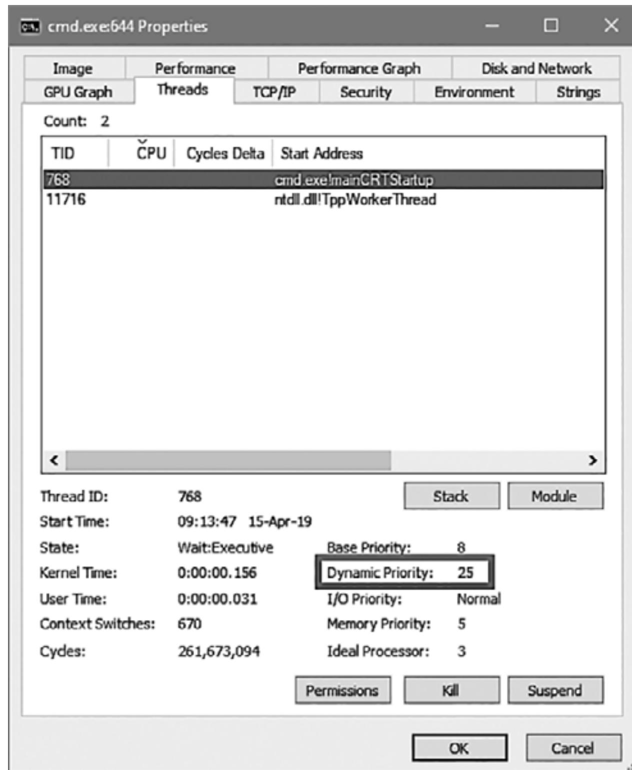


Рис. 4.3. Измененный приоритет потока

Итоги

В этой главе было показано, как создать простой, но вполне функциональный драйвер — от начала и до конца. Мы создали клиент пользовательского режима для взаимодействия с драйвером. Следующая глава будет посвящена отладке — вам непременно придется ею заняться, если написанный вами драйвер ведет себя не так, как ожидалось.

Глава 5
Отладка

Драйверы режима ядра, как и любые другие программы, могут содержать ошибки. Отладка драйверов создает больше проблем по сравнению с отладкой пользовательского режима. Фактически отладка драйвера равнозначна отладке целой машины, не только конкретного процесса или процессов. Она требует другого менталитета. В этой главе рассматривается отладка режима ядра с использованием отладчика WinDbg.

В этой главе:

- Средства отладки для Windows
- Знакомство с WinDbg
- Отладка в режиме ядра
- Полная отладка в режиме ядра
- Введение в отладку драйверов режима ядра

Средства отладки для Windows

Пакет средств отладки для Windows содержит набор отладчиков, служебных программ и документации по отладчикам, входящим в пакет. Этот пакет может устанавливаться в составе Windows SDK или WDK, но никакой реальной «установки» при этом не происходит. Установка просто копирует файлы, но не прикасается к реестру. Таким образом, пакет зависит только от своих модулей и DLL-библиотек Windows. Это позволяет легко скопировать целый каталог в любой другой каталог, в том числе и на съемном носителе.

В пакет входят четыре отладчика: Cdb.exe, Ntsd.exe, Kd.exe и WinDbg.exe. Краткая сводка базовой функциональности каждого отладчика:

- Cdb и Ntsd — консольные отладчики пользовательского режима. Это означает, что они могут присоединяться к процессам, как и любой другой

отладчик пользовательского режима. Оба отладчика имеют консольный пользовательский интерфейс — ввести команду, получить ответ, повторить. Единственное различие между ними заключается в том, что при запуске из консольного окна Cdb использует ту же консоль, тогда как Ntsd открывает новое консольное окно. В остальном они идентичны.

- Kd — отладчик режима ядра с консольным пользовательским интерфейсом. Он может присоединяться как к локальному ядру (локальная отладка ядра, описанная в следующем разделе), так и к другой машине.
- WinDbg — единственный отладчик с графическим интерфейсом. Он может использоваться как при отладке пользовательского режима, так и при отладке режима ядра (в зависимости от выбора команд меню или аргументов командной строки, с которыми он был запущен).

> Новейшая альтернатива для классического отладчика WinDbg — WinDbg Preview — доступна в магазине Microsoft. Это переработанная версия классического отладчика со значительно улучшенным и удобным интерфейсом. WinDbg Preview может устанавливаться в Windows 10 версии 1607 и выше. С точки зрения функциональности он мало отличается от классического WinDbg. Тем не менее он удобнее благодаря современному, пользовательскому интерфейсу и в нем были исправлены некоторые ошибки классического отладчика. Все команды, которые будут приведены позднее в этой главе, нормально работают в обоих отладчиках.

Хотя эти отладки на первый взгляд отличаются друг от друга, в действительности отладчики пользовательского режима работают фактически одинаково, как и отладчики режима ядра. Все они базируются на одном ядре отладки, реализованном в виде DLL-библиотеки (DbgEng.dll). Разные отладчики могут использовать DLL-библиотеки расширения, которые обеспечивают большую часть возможностей отладки.

> Ядро отладки хорошо документировано в документации средств отладки для Windows, что позволяет писать новые отладчики, использующие то же ядро.

В пакет также входят другие служебные программы (неполный список):

- Gflags.exe — программа Global Flags для установки некоторых флагов режима ядра и флагов образов.
- ADPlus.exe — генерирование файла дампа при аварийном завершении или зависании процесса.
- Kill.exe — простая программа для завершения процесса(-ов) по идентификатору процесса, имени или шаблону.

- Dumpchk.exe — программа для общей проверки файлов дампов.
- TList.exe — программа для вывода списка процессов, выполняемых в системе, с различными параметрами.
- Umdh.exe — анализ выделения памяти из кучи в процессах пользовательского режима.
- UsbView.exe — вывод иерархии устройств и концентраторов USB.

Знакомство с WinDbg

В этом разделе изложены основы WinDbg, однако следует помнить, что практически все сказанное в равной степени относится к консольным отладчикам (кроме GUI-окон).

Работа WinDbg строится на основе команд. Пользователь вводит команду, а отладчик отвечает текстом, описывающим результат выполнения команды. С графическим интерфейсом эти результаты отображаются в специальных окнах: локальные переменные, стеки, потоки и т. д.

WinDbg поддерживает три типа команд:

- *Внутренние команды* встроены в отладчик и работают с отлаживаемым объектом.
- *Метакоманды* начинаются с точки (`.`) и работают с самим процессом отладки, а не с непосредственно отлаживаемым объектом.
- *Команды расширения* начинаются с восклицательного знака (`!`) и обеспечивают значительную часть мощи отладчика. Все команды расширения реализуются в DLL-библиотеках расширения. По умолчанию отладчик загружает набор заранее определенных DLL-библиотек расширения, но при необходимости можно загрузить другие библиотеки из каталога отладчика или других источников.

> Написать DLL-библиотеку расширения вполне возможно, этот процесс в полной мере документирован в документации отладчика. Более того, многие такие DLL-библиотеки были созданы и могут быть загружены из соответствующих источников. Эти DLL-библиотеки предоставляют новые команды, расширяющие возможности отладки и часто предназначенные для конкретных сценариев.

Основы отладки пользовательского режима

Если у вас уже есть опыт работы с WinDbg, этот раздел можно спокойно пропустить.

Этот краткий курс дает базовое представление об отладчике WinDbg и о возможностях его применения для отладки пользовательского режима. Отладка режима ядра описана в следующем разделе.

Существуют два основных способа начать отладку пользовательского режима — либо запустить исполняемый файл и присоединить отладчик к нему, либо присоединить его к уже существующему процессу. В этом описании будет использоваться второй вариант, но за исключением первого шага все остальные операции идентичны.

- ✦ Запустите программу Блокнот (Notepad).
- ✦ Запустите WinDbg (Preview или классическую версию. На следующих снимках экрана используется Preview).
- ✦ Выполните команду File ▸ Attach To Process и найдите в списке процесс Notepad (рис. 5.1). Щелкните на кнопке Attach. Примерный результат показан на рис. 5.2.

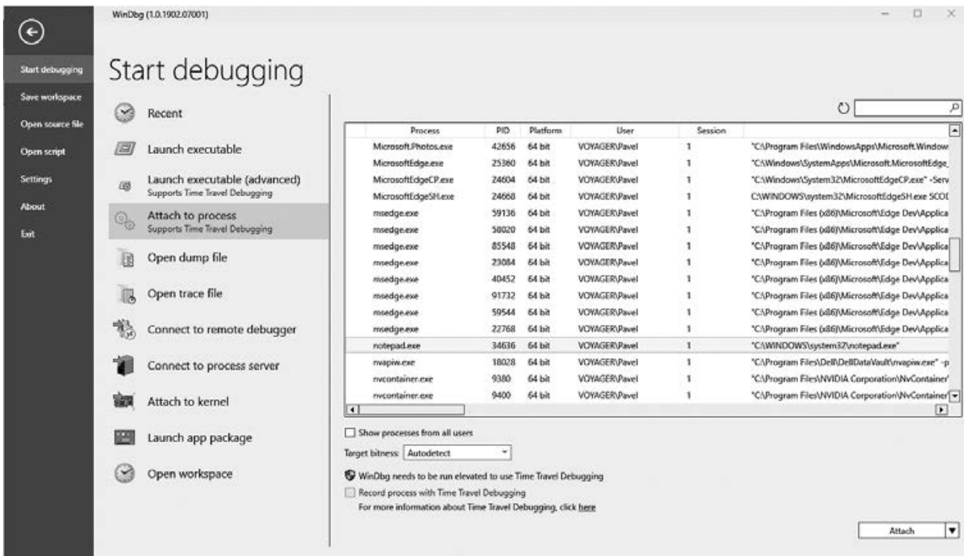


Рис. 5.1. Присоединение к процессу в WinDbg

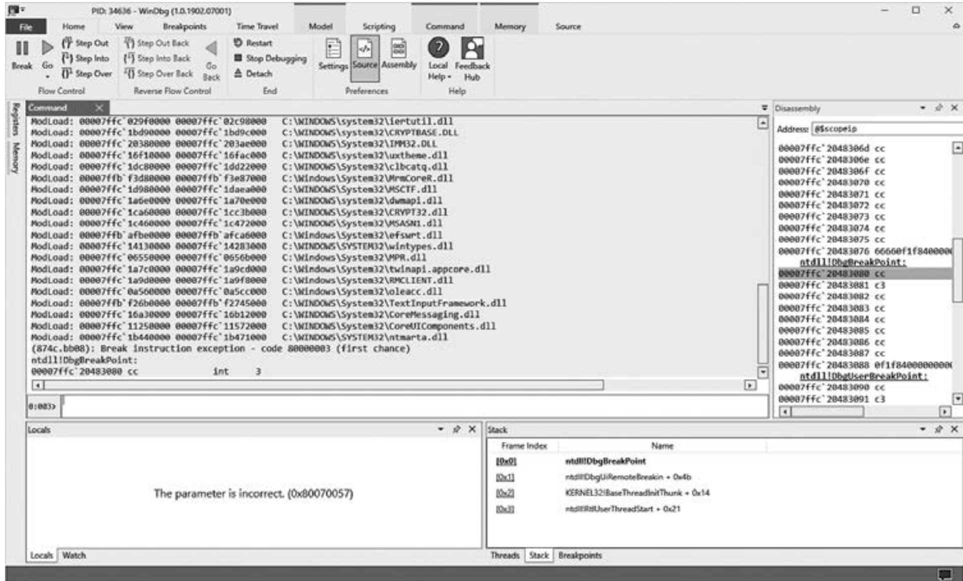


Рис. 5.2. Первое представление после присоединения к процессу

Вас прежде всего интересует окно Command — оно всегда должно оставаться открытым. В нем выводятся результаты различных команд. Обычно большая часть времени в сеансе отладки проводится за взаимодействием с этим окном.

Сейчас процесс приостановлен — вы находитесь в точке прерывания, созданной отладчиком.

✦ Начните с ввода команды ~, которая выводит информацию обо всех потоках в отлаживаемом процессе:

```
0:003> ~
   0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
   1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
   2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
.  3 Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

Точное количество потоков может отличаться от показанного.

Один очень важный аспект отладки — существование символических имен. Компания Microsoft предоставляет общедоступный сервер символических имен, который может использоваться для нахождения символических имен для большинства модулей Microsoft. Символические имена исключительно важны для любой низкоуровневой отладки.

✦ Чтобы настроить символические имена «на скорую руку», введите команду `.symfix`.

- Другой, более правильный вариант — настроить символические имена один раз и предоставить доступ к ним во всех будущих сеансах отладки. Для этого добавьте системную переменную окружения с именем `_NT_SYMBOL_PATH` и присвойте ей строку следующего вида:

```
SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
```

Средняя часть (между звездочками *) содержит локальный путь для кэширования символических имен на вашей локальной машине; вы можете выбрать любой путь по своему усмотрению. После того как значение переменной окружения будет создано, при следующих запусках отладчик будет автоматически находить символические имена и загружать их с сервера символических имен Microsoft по мере надобности.

- Чтобы убедиться в том, что у вас имеются правильные символические имена, введите команду `lm` (Loaded Modules):

```
0:003> lm
start end module name
00007ff7`53820000 00007ff7`53863000 notepad (deferred)
00007ffb`afbe0000 00007ffb`afca6000 efswrt (deferred)

(...)

00007ffc`1db00000 00007ffc`1dba8000 shcore (deferred)
00007ffc`1dbb0000 00007ffc`1dc74000 OLEAUT32 (deferred)
00007ffc`1dc80000 00007ffc`1dd22000 clbcatq (deferred)
00007ffc`1dd30000 00007ffc`1de57000 COMDLG32 (deferred)
00007ffc`1de60000 00007ffc`1f350000 SHELL32 (deferred)
00007ffc`1f500000 00007ffc`1f622000 RPCRT4 (deferred)
00007ffc`1f630000 00007ffc`1f6e3000 KERNEL32 (pdb symbols)     c:\symbols\k\
ernel32.pdb\3B92DED9912D874A2BD08735BC0199A31\kernel32.pdb
00007ffc`1f700000 00007ffc`1f729000 GDI32 (deferred)
00007ffc`1f790000 00007ffc`1f7e2000 SHLWAPI (deferred)
00007ffc`1f8d0000 00007ffc`1f96e000 sechost (deferred)
00007ffc`1f970000 00007ffc`1fc9c000 combase (deferred)
00007ffc`1fca0000 00007ffc`1fd3e000 msvcrt (deferred)
00007ffc`1fe50000 00007ffc`1fef3000 ADVAPI32 (deferred)
00007ffc`20380000 00007ffc`203ae000 IMM32 (deferred)
00007ffc`203e0000 00007ffc`205cd000 ntdll (pdb symbols)       c:\symbols\n\
tdll.pdb\E7EEB80BFAA91532B88FF026DC6B9F341\ntdll.pdb
```

В списке представлены все модули (DLL-библиотеки и EXE), загруженные в отлаживаемом процессе в данный момент времени. Для каждого загруженного модуля указываются начальный и конечный виртуальный адреса. За именем модуля следует описание статуса символических имен модуля (в круглых скобках). Возможные значения:

- `deferred` — символические имена пока не использовались в сеансе отладки, и в данный момент они не загружены. Символические имена будут загружены при необходимости.

- `pdb symbols` — означает, что были загружены правильные открытые символические имена. Далее выводится локальный путь PDB-файла.
- `export symbols` — для DLL-библиотеки доступны только экспортируемые символические имена. Обычно это означает, что для модуля символические имена отсутствуют или не были обнаружены.
- `no symbols` — была сделана попытка найти символические имена модуля, но найти ничего не удалось, даже экспортированные символические имена (такие модули не имеют экспортированных символических имен, как исполняемые файлы и файлы драйверов).

Принудительная загрузка символических имен модуля выполняется командой `.reload /f modulename.dll`. Такая команда может дать убедительные доказательства наличия символических имен для данного модуля.

Пути к символическим именам также можно настроить в диалоговом окне настроек отладчика.

- Откройте меню File ▸ Settings и найдите раздел Debugging Settings. В нем можно добавить дополнительные пути для поиска символических имен. Например, это может быть полезно при отладке вашего кода, чтобы отладчик просматривал ваши каталоги, в которых могут быть найдены соответствующие PDB-файлы (рис. 5.3).

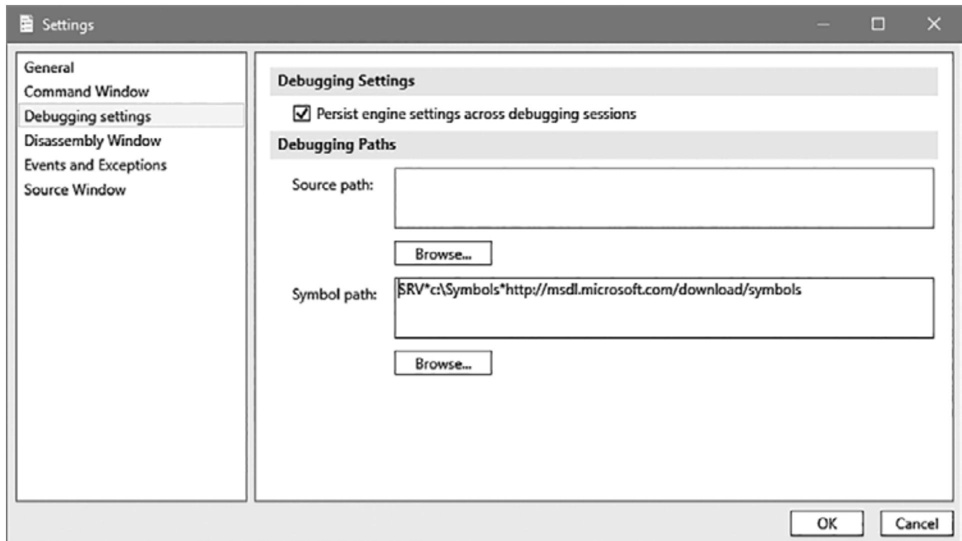


Рис. 5.3. Настройка путей к символическим именам и исходному коду

✦ Прежде чем продолжать, проверьте правильность символических имен. Чтобы выявить любые проблемы, воспользуйтесь командой `!sym noisy`, которая записывает в журнал подробную информацию о попытках загрузки символических имен.

Вернемся к списку потоков — обратите внимание на то, что у одного из потоков перед данными стоит точка. Этот поток является текущим с точки зрения отладчика. Это означает, что любая введенная команда для потока, в которой поток явно не задан, будет выполнена с этим потоком. «Текущий поток» также показан в приглашении — число справа от двоеточия является индексом текущего потока (3 в данном примере).

✦ Введите команду `k`, которая выводит трассировку стека текущего потока:

```
0:003> k
 # Child-SP          RetAddr           Call Site
00 00000001`224ffbd8 00007ffc`204aef5b ntdll!DbgBreakPoint
01 00000001`224ffbe0 00007ffc`1f647974 ntdll!DbgUiRemoteBreakin+0x4b
02 00000001`224ffc10 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`224ffc40 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

В трассировке приведен список вызовов в этом потоке (конечно, только в пользовательском режиме). На вершине стека в приведенном выводе находится функция `DbgBreakPoint`, находящаяся в модуле ntdll.dll. Адреса с символическими именами имеют общий формат *имя_модуля*!*имя_функции*+*смещение*. Смещение не является обязательным и может быть равно нулю, если вызов относится к самому началу функции. Также обратите внимание на то, что имя модуля указывается без расширения.

В приведенном выводе функция `DbgBreakpoint` вызывается `DbgUiRemoteBreakIn`, которая была вызвана функцией `BaseThreadInitThunk` и т. д.

> Кстати говоря, отладчик внедряет код в этот поток для принудительного прерывания выполнения.

✦ Чтобы переключиться на другой поток, введите команду `~ns`, где `n` — индекс потока. Переключимся на поток 0 и выведем его стек вызовов:

```
0:003> ~0s
win32u!NtUserGetMessage+0x14:
00007ffc`1c4b1164 c3              ret
0:000> k
 # Child-SP          RetAddr           Call Site
00 00000001`2247f998 00007ffc`1d802fbd win32u!NtUserGetMessage+0x14
01 00000001`2247f9a0 00007ff7`5382449f USER32!GetMessageW+0x2d
02 00000001`2247fa00 00007ff7`5383ae07 notepad!WinMain+0x267
03 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x19f
04 00000001`2247fbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
05 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Это главный (первый) поток Notepad. В начале стека показан поток, ожидающий UI-сообщений.

Альтернативный способ вывода стека вызовов для другого потока без переключения на него — включение символа ~ и номера потока перед командой. В следующем фрагменте выводится стек для потока 1:

```
0:000> ~1k
 # Child-SP          RetAddr           Call Site
00 00000001`2267f4c8 00007ffc`204301f4 ntdll!NtWaitForWorkViaWorkerFactory+0x14
01 00000001`2267f4d0 00007ffc`1f647974 ntdll!TppWorkerThread+0x274
02 00000001`2267f7c0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`2267f7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Вернемся к списку потоков:

```
. 0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
  1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
  2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
# 3 Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

Обратите внимание: точка переместилась к потоку 0 (текущий поток), а у потока 3 появился символ #. Этим символом помечается поток, который стал причиной точки прерывания (в нашем случае это исходный поток, к которому присоединился отладчик).

Базовая информация о потоке выводится командой ~ (рис. 5.4).

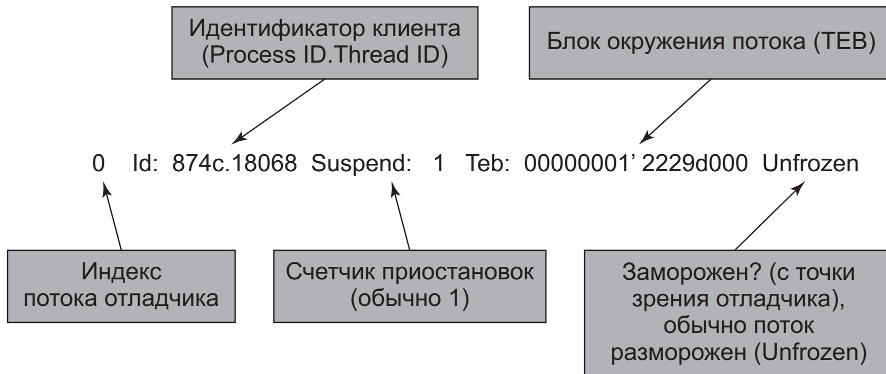


Рис. 5.4. Информация о потоке, выводимая командой ~

WinDbg выводит большинство чисел в шестнадцатеричной системе. Чтобы преобразовать значение в десятичную запись, воспользуйтесь командой ? (вычисление выражения).

- ✦ Введите следующую команду для получения десятичного идентификатора процесса (сравните со значением PID в диспетчере задач):

```
0:000> ? 874c
Evaluate expression: 34636 = 00000000`0000874c
```

- Десятичный формат также выражается префиксом `0n`, поэтому вы можете провести обратное преобразование:

```
0:000> ? 0n34636
Evaluate expression: 34636 = 00000000`0000874c
```

- Команда `!teb` используется для просмотра блока TEB потока. При выполнении команды `!teb` без адреса выводится TEB текущего потока:

```
0:000> !teb
TEB at 000000012229d000
    ExceptionList:        0000000000000000
    StackBase:            0000000122480000
    StackLimit:           000000012246f000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000000012229d000
    EnvironmentPointer:   0000000000000000
    ClientId:             000000000000874c . 0000000000018068
    RpcHandle:            0000000000000000
    Tls Storage:          000001c93676c940
    PEB Address:          000000012229c000
    LastErrorValue:       0
    LastStatusValue:      8000001a
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> !teb 00000001`222a5000
TEB at 00000001222a5000
    ExceptionList:        0000000000000000
    StackBase:            0000000122680000
    StackLimit:           000000012266f000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 00000001222a5000
    EnvironmentPointer:   0000000000000000
    ClientId:             000000000000874c . 00000000000046ac
    RpcHandle:            0000000000000000
    Tls Storage:          000001c936764260
    PEB Address:          000000012229c000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0
```

Некоторые данные, выводимые командой `!teb`, относительно хорошо известны:

- `StackBase` и `StackLimit` — база стека пользовательского режима и ограничение потока.

- ClientId — идентификаторы процесса и потока.
- LastErrorValue — последний код ошибки Win32 (GetLastError).
- TlsStorage — массив TLS (Thread Local Storage) для этого потока (полное объяснение TLS выходит за рамки книги).
- PEB Address — адрес блока PEB (Process Environment Block), для просмотра которого можно воспользоваться командой !peb.

Команда !teb (и другие аналогичные команды) выводит части реальной структуры (в данном случае _TEB). Для просмотра реальной структуры можно воспользоваться командой dt (Display Type):

```
0:000> dt ntdll!_teb
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : Ptr64 Void
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : Ptr64 Void
   +0x058 ThreadLocalStoragePointer : Ptr64 Void
   +0x060 ProcessEnvironmentBlock : Ptr64 _PEB

(...)

   +0x1808 LockCount        : Uint4B
   +0x180c WowTebOffset     : Int4B
   +0x1810 ResourceRetValue : Ptr64 Void
   +0x1818 ReservedForWdf   : Ptr64 Void
   +0x1820 ReservedForCrt   : Uint8B
   +0x1828 EffectiveContainerId : _GUID
```

Следует заметить, что WinDbg не различает регистр символов в символических именах. Также обратите внимание на то, что имя структуры начинается с символа подчеркивания: так определяются все структуры Windows (как в пользовательском режиме, так и в режиме ядра). Использование имени без подчеркивания может работать, но может и не работать, поэтому я рекомендую всегда включать символ подчеркивания.

Как определить, в каком модуле определяется просматриваемая структура? Если структура документирована, то модуль будет указан в документации структуры. Также можно попытаться указать структуру без имени модуля, чтобы отладчик попытался найти ее. Обычно по опыту (а иногда и по контексту) разработчик уверенно может предположить, где определяется структура.

- Присоединив адрес к приведенной выше команде, можно просмотреть реальные значения полей данных:

```
0:000> dt ntdll!_teb 00000001`2229d000
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
```

```
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : 0x000001c9`3676c940 Void
   +0x060 ProcessEnvironmentBlock : 0x00000001`2229c000 _PEB
   +0x068 LastErrorValue   : 0

(...)

   +0x1808 LockCount        : 0
   +0x180c WowTebOffset     : 0n0
   +0x1810 ResourceRetValue : 0x000001c9`3677fd00 Void
   +0x1818 ReservedForWdf   : (null)
   +0x1820 ReservedForCrt   : 0
   +0x1828 EffectiveContainerId : _GUID {00000000-0000-0000-0000-000000000000}
```

Для каждого поля указывается смещение от начала структуры, имя и значение. Простые значения выводятся напрямую, тогда как значения-структуры (как, например, `NtTib` в этом фрагменте) обычно выводятся в виде гиперссылки. Эта гиперссылка открывает подробное описание структуры.

✦ Щелкните на поле `NtTib`, чтобы просмотреть подробную информацию о поле данных:

```
0:000> dx -r1 (*((ntdll!_NT_TIB *)0x12229d000))
(*((ntdll!_NT_TIB *)0x12229d000)) [Type: _NT_TIB]
    [+0x000] ExceptionList   : 0x0 [Type: _EXCEPTION_REGISTRATION_RECORD *]
    [+0x008] StackBase       : 0x122480000 [Type: void *]
    [+0x010] StackLimit      : 0x12246f000 [Type: void *]
    [+0x018] SubSystemTib    : 0x0 [Type: void *]
    [+0x020] FiberData       : 0x1e00 [Type: void *]
    [+0x020] Version         : 0x1e00 [Type: unsigned long]
    [+0x028] ArbitraryUserPointer : 0x0 [Type: void *]
    [+0x030] Self            : 0x12229d000 [Type: _NT_TIB *]
```

Для просмотра данных в отладчике используется более новая команда `dx`.

> Если гиперссылки не отображаются, возможно, вы используете очень старую версию WinDbg, в которой язык DML (Debugger Markup Language) не включен по умолчанию. Он включается командой `.prefer_dml 1`.

Теперь обратимся к точкам прерывания. Установим точку прерывания при открытии файла в Блокноте.

Введите следующую команду, чтобы установить точку прерывания в функции API `CreateFile`:

```
0:000> bp kernel32!createfilew
```

Обратите внимание на имя функции `CreateFileW`: функции с именем `CreateFile` не существует. В программном коде это имя представляет макрос, который

расширяется в `CreateFileW` («широкая» версия для Юникода) или `CreateFileA` (версия для ASCII или ANSI) в зависимости от константы компиляции с именем `UNICODE`. WinDbg никак не реагирует на команду, и это хороший знак.

> Большинство строковых функций API существует в двух версиях по историческим причинам. В любом случае проекты Visual Studio по умолчанию определяют константу UNICODE, так что Юникод является нормой. И это хорошо — A-функции преобразуют свои входные данные в Юникод и вызывают W-функции.

✦ Для просмотра существующих точек прерывания можно воспользоваться командой `bl`:

```
0:000> bl
 0 e Disable Clear 00007ffc`1f652300 0001 (0001) 0:**** KERNEL32!CreateFileW
```

В результатах указан индекс точки прерывания (0), ее состояние (`e` = активна, `d` = заблокирована), а также включаются гиперссылки для блокировки (команда `bd`) и удаления (команда `bc`) точек прерывания.

Продолжим выполнение Блокнота, пока не будет достигнута точка прерывания:

✦ Введите команду `g`, нажмите кнопку Go на панели инструментов или нажмите клавишу F5.

В приглашении отладчика выводится сообщение `Busy`, а в области команд — сообщение `Debuggee is running`; это означает, что вы не сможете вводить команды до следующего прерывания.

✦ Процесс Notepad должен работать. Откройте меню File и выберите команду Open.... Отладчик должен выдать подробную информацию о загрузке модулей, а затем прервать выполнение программы:

```
Breakpoint 0 hit
KERNEL32!CreateFileW:
00007ffc`1f652300 ff25aa670500 jmp qword ptr [KERNEL32!_imp_CreateFileW (0000\
7ffc`1f6a8ab0)] ds:00007ffc`1f6a8ab0={KERNELBASE!CreateFileW (00007ffc`1c75e260)}
```

✦ Точка прерывания достигнута! Обратите внимание на поток, в котором это произошло. Посмотрим, как выглядит стек вызовов (возможно, информация появится не сразу, если отладчику понадобится загрузить символические имена с сервера символических имен Microsoft):

```
0:002> k
 # Child-SP          RetAddr           Call Site
00 00000001`226fab08 00007ffc`061c8368 KERNEL32!CreateFileW
01 00000001`226fab10 00007ffc`061c5d4d mscoreei!RuntimeDesc::VerifyMainRuntimeMod
ule\
+0x2c
```

```
02 00000001`226fab60 00007ffc`061c6068 mscoreei!FindRuntimesInInstallRoot+0x2fb
03 00000001`226fb3e0 00007ffc`061cb748 mscoreei!GetOrCreateSxSProcessInfo+0x94
04 00000001`226fb460 00007ffc`061cb62b mscoreei!CLRMetaHostPolicyImpl::GetRequest
edR\
untimeHelper+0xfc
05 00000001`226fb740 00007ffc`061ed4e6 mscoreei!CLRMetaHostPolicyImpl::GetRequest
edR\
untime+0x120

(...)

21 00000001`226fede0 00007ffc`1df025b2 SHELL32!CFSIconOverlayManager::LoadNonload
edO\
verlayIdentifiers+0xaa
22 00000001`226ff320 00007ffc`1df022af SHELL32!EnableExternalOverlayIdentifiers+
0x46
23 00000001`226ff350 00007ffc`1def434e SHELL32!CFSIconOverlayManager::RefreshOver
lay\
Images+0xff
24 00000001`226ff390 00007ffc`1cf250a3 SHELL32!SHELL32_GetIconOverlayManager+0x6e
25 00000001`226ff3c0 00007ffc`1ceb2726 windows_storage!CFSFolder::_
GetOverlayInfo+0x\
12b
26 00000001`226ff470 00007ffc`1cf3108b windows_storage!CAutoDestItemsFolder::GetO
ver\
layIndex+0xb6
27 00000001`226ff4f0 00007ffc`1cf30f87 windows_storage!CRegFolder::_
GetOverlayInfo+0\
xbf
28 00000001`226ff5c0 00007ffb`df8fc4d1 windows_storage!CRegFolder::GetOverlayInd
ex+0\
x47
29 00000001`226ff5f0 00007ffb`df91f095 explorerframe!CNscOverlayTask::_
Extract+0x51
2a 00000001`226ff640 00007ffb`df8f70c2 explorerframe!CNscOverlayTask::InternalRes
ume\
RT+0x45
2b 00000001`226ff670 00007ffc`1cf7b58c explorerframe!CRunnableTask::Run+0xb2
2c 00000001`226ff6b0 00007ffc`1cf7b245 windows_storage!CShellTask::TT_Run+0x3c
2d 00000001`226ff6e0 00007ffc`1cf7b125 windows_storage!CShellTaskThread::ThreadPr
oc+\
0xdd
2e 00000001`226ff790 00007ffc`1db32ac6 windows_storage!CShellTaskThread::s_
ThreadPro\
c+0x35
2f 00000001`226ff7c0 00007ffc`204521c5 shcore!ExecuteWorkItemThreadProc+0x16
30 00000001`226ff7f0 00007ffc`204305c4 ntdll!RtlpTpWorkCallback+0x165
31 00000001`226ff8d0 00007ffc`1f647974 ntdll!TppWorkerThread+0x644
32 00000001`226ffbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
33 00000001`226ffbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Что можно сделать в этой точке? Можно поинтересоваться, какой файл был открыт. Эту информацию можно получить на основании конвенции вызова

функции `CreateFileW`. Так как процесс является 64-разрядным (и используется процессор Intel/AMD), конвенция вызова указывает, что первые целочисленные аргументы/указатели передаются в регистрах RCX, RDX, R8 и R9. Так как имя файла передается `CreateFileW` в первом аргументе, оно хранится в регистре RCX.

> Дополнительная информация о конвенциях вызова приведена в документации отладчика (а также в Сети).

- Выведите значение регистра RCX командой `r` (вы получите другое значение):

```
0:002> r rcx
rcx=00000001226fabf8
```

- Для просмотра памяти, на которую указывает RCX, можно воспользоваться командой `d` (Display):

```
0:002> db 00000001226fabf8
00000001`226fabf8 43 00 3a 00 5c 00 57 00-69 00 6e 00 64 00 6f 00 C.:.\.W.i.n.d.o.
00000001`226fac08 77 00 73 00 5c 00 4d 00-69 00 63 00 72 00 6f 00 w.s.\.M.i.c.r.o.
00000001`226fac18 73 00 6f 00 66 00 74 00-2e 00 4e 00 45 00 54 00 s.o.f.t...N.E.T.
00000001`226fac28 5c 00 46 00 72 00 61 00-6d 00 65 00 77 00 6f 00 \.F.r.a.m.e.w.o.
00000001`226fac38 72 00 6b 00 36 00 34 00-5c 00 5c 00 76 00 32 00 r.k.6.4.\.\.v.2.
00000001`226fac48 2e 00 30 00 2e 00 35 00-30 00 37 00 32 00 37 00 ..0...5.0.7.2.7.
00000001`226fac58 5c 00 63 00 6c 00 72 00-2e 00 64 00 6c 00 6c 00 \.c.l.r...d.l.l.
00000001`226fac68 00 00 76 1c fc 7f 00 00-00 00 00 00 00 00 00 00 ..v.............
```

Команда `db` выводит содержимое памяти в виде байтов; справа приводятся ASCII-символы. Понять имя файла можно, но из-за того, что строка хранится в Юникоде, просматривать ее неудобно.

- Команда `du` позволяет просматривать строки Юникода в более удобном виде:

```
0:002> du 00000001226fabf8
00000001`226fabf8 "C:\Windows\Microsoft.NET\Framewo"
00000001`226fac38 "rk64\\v2.0.50727\clr.dll"
```

- Чтобы просмотреть значение регистра напрямую, поставьте перед его именем префикс `@`:

```
0:002> du @rcx
00000001`226fabf8 "C:\Windows\Microsoft.NET\Framewo"
00000001`226fac38 "rk64\\v2.0.50727\clr.dll"
```

Теперь установим другую точку прерывания в платформенной функции API, которая вызывается из `CreateFileW` — `NtCreateFile`:

```
0:002> bp ntdll!ntcreatefile
0:002> bl
```

```
  0 e Disable Clear 00007ffc`1f652300 0001 (0001) 0:****   KERNEL32!CreateFileW
  1 e Disable Clear 00007ffc`20480120 0001 (0001) 0:**** ntdll!NtCreateFile
```

Обратите внимание: платформенные функции API никогда не используют версии `W` или `A` — они всегда работают со строками в Юникоде.

✦ Продолжите выполнение командой `g`. Отладчик прерывает выполнение программы:

```
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1         mov r10,rcx
```

✦ Снова проверьте состояние стека вызовов:

```
0:002> k
# Child-SP           RetAddr            Call Site
00 00000001`226fa938 00007ffc`1c75e5d6 ntdll!NtCreateFile
01 00000001`226fa940 00007ffc`1c75e2c6 KERNELBASE!CreateFileInternal+0x2f6
02 00000001`226faab0 00007ffc`061c8368 KERNELBASE!CreateFileW+0x66
03 00000001`226fab10 00007ffc`061c5d4d mscoreei!RuntimeDesc::VerifyMainRuntimeMod
ule\
+0x2c
04 00000001`226fab60 00007ffc`061c6068 mscoreei!FindRuntimesInInstallRoot+0x2fb
05 00000001`226fb3e0 00007ffc`061cb748 mscoreei!GetOrCreateSxSProcessInfo+0x94

(...)
```

Команда `u` (Unassemble) выводит следующие 8 команд, которые будут выполнены:

```
0:002> u
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1          mov r10,rcx
00007ffc`20480123 b855000000      mov eax,55h
00007ffc`20480128 f604250803fe7f01 test byte ptr
                                  [SharedUserData+0x308 (00000000`\
7ffe0308)],1
00007ffc`20480130 7503            jne ntdll!NtCreateFile+0x15 (00007ffc`20480135)
00007ffc`20480132                 0f05 syscall
00007ffc`20480134                 c3 ret
00007ffc`20480135                 cd2e int 2Eh
00007ffc`20480137                 c3 ret
```

Обратите внимание на команду, копирующую значение 0x55 в регистр EAX. Это номер системной сервисной функции `NtCreateFile` (см. главу 1). Команда `syscall` осуществляет переход в режим ядра для выполнения системной сервисной функции `NtCreateFile`.

✦ Команда `p` осуществляет пошаговое выполнение следующей команды (также можно нажать клавишу F10). Для пошагового выполнения с заходом

в функцию (в случае сборки это команда `call`) используется команда `t` (альтернатива — нажатие клавиши F11).

```
0:002> p
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1          mov r10,rcx
0:002> p
ntdll!NtCreateFile+0x3:
00007ffc`20480123 b855000000      mov eax,55h
0:002> p
ntdll!NtCreateFile+0x8:
00007ffc`20480128 f604250803fe7f01 test byte ptr [SharedUserData+0x308
(00000000`\
7ffe0308)],1 ds:00000000`7ffe0308=00
0:002> p
ntdll!NtCreateFile+0x10:
00007ffc`20480130 7503            jne ntdll!NtCreateFile+0x15 (00007ffc`20480135\
) [br=0]
0:002> p
ntdll!NtCreateFile+0x12:
00007ffc`20480132 0f05            syscall
```

- Пошаговое выполнение с заходом в `syscall` невозможно, так как мы находимся в пользовательском режиме. При пошаговом выполнении с обходом функция выполняется немедленно, и мы получаем результат.

```
0:002> p
ntdll!NtCreateFile+0x14:
00007ffc`20480134 c3              ret
```

- Возвращаемое значение функции в конвенции вызова x64 хранится в EAX или RAX. Для системных вызовов это значение `NTSTATUS`, поэтому возвращаемый статус содержится в регистре EAX:

```
0:002> r eax
eax=c0000034
```

- Произошла ошибка. Для получения подробной информации можно воспользоваться командой `!error`:

```
0:002> !error @eax
Error code: (NTSTATUS) 0xc0000034 (3221225524) - Object Name not found.
```

- Заблокируйте все точки прерывания и продолжите нормальное выполнение Notepad:

```
0:002> bd *
0:002> g
```

Так как на данный момент точки прерывания отсутствуют, можно принудительно прервать выполнение кнопкой Break на панели инструментов или нажатием клавиш Ctrl+Break:

```
874c.16a54): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffc`20483080                 cc int 3
```

Обратите внимание на номер потока в приглашении. Выведите список всех текущих потоков:

```
0:022> ~
   0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
   1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
   2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
   3 Id: 874c.f7ec Suspend: 1 Teb: 00000001`222ad000 Unfrozen
   4 Id: 874c.145b4 Suspend: 1 Teb: 00000001`222af000 Unfrozen

(...)

   18 Id: 874c.f0c4 Suspend: 1 Teb: 00000001`222d1000 Unfrozen
   19 Id: 874c.17414 Suspend: 1 Teb: 00000001`222d3000 Unfrozen
   20 Id: 874c.c878 Suspend: 1 Teb: 00000001`222d5000 Unfrozen
   21 Id: 874c.d8c0 Suspend: 1 Teb: 00000001`222d7000 Unfrozen
 . 22 Id: 874c.16a54 Suspend: 1 Teb: 00000001`222e1000 Unfrozen
   23 Id: 874c.10838 Suspend: 1 Teb: 00000001`222db000 Unfrozen
   24 Id: 874c.10cf0 Suspend: 1 Teb: 00000001`222dd000 Unfrozen
```

Многовато потоков, не так ли? Они были созданы/активизированы стандартным диалоговым окном открытия файла, так что вины Notepad в этом нет.

✦ Продолжайте экспериментировать с отладчиком.

Найдите номера системных сервисных функций для NtWriteFile и NtReadFile.

✦ При закрытии Блокнота сработает точка прерывания при завершении процесса:

```
ntdll!NtTerminateProcess+0x14:

00007ffc`2047fc14 c3             ret
0:000> k
 # Child-SP          RetAddr           Call Site
00 00000001`2247f6a8 00007ffc`20446dd8 ntdll!NtTerminateProcess+0x14
01 00000001`2247f6b0 00007ffc`1f64d62a ntdll!RtlExitUserProcess+0xb8
02 00000001`2247f6e0 00007ffc`061cee58 KERNEL32!ExitProcessImplementation+0xa
03 00000001`2247f710 00007ffc`0644719e mscoreei!RuntimeDesc::ShutdownAllActiveRun
tim\
es+0x287
04 00000001`2247fa00 00007ffc`1fcda291 mscoree!ShellShim_CorExitProcess+0x11e
05 00000001`2247fa30 00007ffc`1fcda2ad msvcrt!_crtCorExitProcess+0x4d
```

```
06 00000001`2247fa60 00007ffc`1fcda925 msvcrt!_crtExitProcess+0xd
07 00000001`2247fa90 00007ff7`5383ae1e msvcrt!doexit+0x171
08 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x1b6
09 00000001`2247fbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
0a 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

- Введите команду `q`, чтобы выйти из отладчика. Если процесс еще работает, он будет завершен. Также можно ввести команду `.detach` для отсоединения от целевого потока без его уничтожения.

Отладка режима ядра

В ходе отладки пользовательского режима отладчик присоединяется к процессу, устанавливает точки прерывания, которые вызывают приостановку потока процесса, и т. д. С другой стороны, отладка режима ядра основана на управлении всей машиной из отладчика. Это означает, что при установке и последующем срабатывании точки прерывания блокируется вся машина. Очевидно, на одной машине это невозможно. В полноценной отладке ядра участвуют две машины: хост (на котором работает отладчик) и управляемая машина (на которой работает программа). Впрочем, управляемая машина может быть виртуальной, находящейся на той же машине (хосте), на которой работает отладчик. На рис. 5.5 изображены хост и управляемая машина, соединенные некоторым каналом связи.

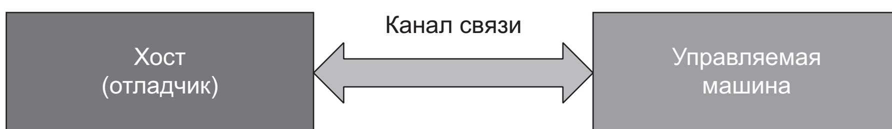


Рис. 5.5. Связь хоста с управляемой машиной

Но прежде чем браться за полную отладку режима ядра, рассмотрим ее упрощенную разновидность — локальную отладку режима ядра.

Локальная отладка режима ядра

Локальная отладка режима ядра (LKD, Local Kernel Debugging) позволяет просматривать системную память и другую системную информацию на локальной машине. Основное различие между локальной и полной отладкой режима ядра заключается в том, что в LKD отсутствует возможность установки точек прерывания; таким образом, вы всегда просматриваете текущее состояние системы. Также это означает, что ситуация изменяется даже во время выпол-

нения команд, так что часть информации может оказаться ненадежной. При полной отладке ядра команды могут вводиться только в то время, пока целевая система находится в точке прерывания, так что состояние системы остается неизменным.

Чтобы настроить LKD, введите следующую команду в окне командной строки с повышенными привилегиями, после чего перезапустите систему:

```
bcdedit /debug on
```

После того как система будет перезагружена, запустите WinDbg с повышенными привилегиями. Выберите команду меню File ▸ Attach To Kernel (WinDbg Preview) или File ▸ Kernel Debug... (классический WinDbg). Перейдите на вкладку Local и щелкните на кнопке OK. Результат должен выглядеть примерно так:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Connected to Windows 10 18362 x64 target at (Sun Apr 21 08:50:59.964 2019 (UTC +
3:0\
0)), ptr64 TRUE

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       SRV*c:\Symbols*http://msdl.
microsoft.\
com/download/symbols
Symbol search path is: c:\temp;SRV*c:\Symbols*http://msdl.microsoft.com/download/
sym\
bols
Executable search path is:
Windows 10 Kernel Version 18362 MP (12 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff806`466b8000 PsLoadedModuleList = 0xfffff806`46afb2d0
Debug session time: Sun Apr 21 08:51:00.702 2019 (UTC + 3:00)
System Uptime: 0 days 11:33:37.265
```

Локальная отладка ядра защищается механизмом Secure Boot в Windows 10, Server 2016 и последующих версий. Чтобы активизировать LKD, необходимо отключить Secure Boot в настройках BIOS машины. Если по какой-то причине это невозможно, существует альтернативное решение с использованием программы LiveKd из пакета Sysinternals. Скопируйте LiveKd.exe в главный каталог средств отладки Windows. Затем запустите WinDbg с использованием LiveKd следующей командой: `livekd -w`.

В приглашении выводится текст *lkd*. Он означает, что локальная отладка режима ядра активна.

Знакомство с локальной отладкой режима ядра

Если вы знакомы с основными командами отладки режима ядра, этот раздел можно пропустить.

Базовая информация обо всех процессах, работающих в системе, выводится командой `process 0 0`:

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffff8d0e682a73c0
    SessionId: none Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase:  001ad002  ObjectTable: ffffe20712204b80  HandleCount: 9542.
    Image: System

PROCESS ffff8d0e6832e140
    SessionId: none Cid: 0058    Peb: 00000000  ParentCid: 0004
    DirBase:  03188002  ObjectTable: ffffe2071220cac0  HandleCount: 0.
    Image: Secure System

PROCESS ffff8d0e683f1080
    SessionId: none Cid: 0098    Peb: 00000000  ParentCid: 0004
    DirBase:  003e1002  ObjectTable: ffffe20712209480  HandleCount: 0.
    Image: Registry

PROCESS ffff8d0e83099080
    SessionId: none Cid: 032c     Peb: 5aba7eb000  ParentCid: 0004
    DirBase:  15fa39002  ObjectTable: ffffe20712970080  HandleCount: 53.
    Image: smss.exe

(...)
```

Для каждого процесса выводится следующая информация:

- Адрес после текста `PROCESS` — адрес `EPROCESS` процесса (в пространстве ядра, конечно).
- `SessionId` — сеанс, в котором выполняется процесс.
- `Cid` — (идентификатор клиента) уникальный идентификатор процесса.
- `Peb` — адрес блока PEB (Process Environment Block). Естественно, этот адрес относится к пространству пользовательского режима.
- `ParentCid` — (идентификатор родителя клиента) идентификатор родительского процесса. Существует некоторая вероятность того, что родительский процесс уже не существует, и идентификатор может быть использован повторно.
- `DirBase` — физический адрес (без младших 12 разрядов) главного каталога страниц (Master Page Directory) для этого процесса, используемого в качестве базы для преобразования виртуальных адресов. На платформе

x64 используется термин «карта страниц 4-го уровня» (Page Map Level 4), а на платформе x86 — «таблица указателей каталога страниц» (PDPT, Page Directory Pointer Table).

- `ObjectTable` — указатель на таблицу приватных дескрипторов процесса.
- `HandleCount` — количество дескрипторов в процессе.
- `Image` — имя исполняемого файла или специальное имя процесса, не связанного с исполняемым файлом (примеры: Secure System, System, Mem Compression).

Команда `!process` получает как минимум два аргумента. Первый аргумент обозначает процесс (задается адресом `EPROCESS`), нулевое значение обозначает «все или любые процессы». Второй аргумент определяет нужный уровень детализации, 0 обозначает наименьшее количество подробностей (битовая маска). Третий аргумент может быть добавлен для поиска конкретного исполняемого файла.

- Выведите список всех процессов, в которых выполняется csrss.exe:

```
lkd> !process 0 0 csrss.exe
PROCESS ffff8d0e83c020c0
    SessionId: 0  Cid: 038c    Peb: f599af6000  ParentCid: 0384
    DirBase: 844eaa002 ObjectTable: ffffe20712345480 HandleCount: 992.
    Image: csrss.exe

PROCESS ffff8d0e849df080
    SessionId: 1  Cid: 045c    Peb: e8a8c9c000  ParentCid: 0438
    DirBase: 17afc1002 ObjectTable: ffffe207186d93c0 HandleCount: 1146.
    Image: csrss.exe
```

- Выведите расширенную информацию по конкретному процессу, указав его адрес и более высокий уровень детализации:

```
lkd> !process ffff8d0e849df080 1
PROCESS ffff8d0e849df080
    SessionId: 1  Cid: 045c    Peb: e8a8c9c000  ParentCid: 0438
    DirBase: 17afc1002 ObjectTable: ffffe207186d93c0 HandleCount: 1138.
    Image: csrss.exe
    VadRoot ffff8d0e999a4840 Vads 244 Clone 0 Private 670. Modified 48241. Locked
38\
106.
    DeviceMap ffffe20712213720
    Token                             ffffe207186f38f0
    ElapsedTime                       12:14:47.292
    UserTime                          00:00:00.000
    KernelTime                        00:00:03.468
    QuotaPoolUsage[PagedPool]         423704
    QuotaPoolUsage[NonPagedPool]      37752
    Working Set Sizes (now,min,max) (1543, 50, 345) (6172KB, 200KB, 1380KB)
    PeakWorkingSetSize                10222
```

```
    VirtualSize                       2101434 Mb
    PeakVirtualSize                   2101467 Mb
    PageFaultCount                    841489
    MemoryPriority                    BACKGROUND
    BasePriority                      13
    CommitCharge                      1012
    Job                               ffff8d0e83da8080
```

Как видно из результатов, эта команда выводит более подробную информацию о процессе. Часть этой информации оформлена в виде гиперссылок для дальнейшего анализа данных. Задание, частью которого является процесс (если оно есть), представлено гиперссылкой.

✦ Щелкните на гиперссылке с адресом задания Job:

```
lkd> !job ffff8d0e83da8080
Job at ffff8d0e83da8080
  Basic Accounting Information
    TotalUserTime:             0x33db258
    TotalKernelTime:           0x5705d50
    TotalCycleTime:            0x73336f9ae
    ThisPeriodTotalUserTime:   0x33db258
    ThisPeriodTotalKernelTime: 0x5705d50
    TotalPageFaultCount:       0x8617c
    TotalProcesses:            0x3e
    ActiveProcesses:           0xd
    FreezeCount:               0
    BackgroundCount:           0
    TotalTerminatedProcesses:  0x0
    PeakJobMemoryUsed:         0x38fb5
    PeakProcessMemoryUsed:     0x29366
  Job Flags
    [wake notification allocated]
    [wake notification enabled]
    [timers virtualized]
  Limit Information (LimitFlags: 0x1800)
  Limit Information (EffectiveLimitFlags: 0x1800)
    JOB_OBJECT_LIMIT_BREAKAWAY_OK
    JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK
```

> Задание представляет собой объект, содержащий один или несколько процессов, к которым могут применяться различные ограничения и отслеживаться различная учетная информация. Подробное обсуждение заданий выходит за рамки книги. За дополнительной информацией обращайтесь к книгам серии «Внутреннее устройство Windows».

✦ Как обычно, такая команда, как `!job`, скрывает часть информации, доступной в реальной структуре данных. В данном случае это структура `EJOB`. Чтобы просмотреть все подробности, используйте команду `dt nt!_ejob` с адресом задания.

✦ Также вы можете просмотреть блок PEB процесса, щелкнув на его гиперссылке. Результат выглядит примерно так же, как при выполнении команды `!peb` в пользовательском режиме, но здесь есть один нюанс: сначала необходимо задать правильный контекст процесса, так как адрес находится в пользовательском режиме. Щелкните на гиперссылке Peb. Результат должен выглядеть примерно так:

```
lkd> .process /p ffff8d0e849df080; !peb e8a8c9c000
Implicit process is now ffff8d0e`849df080
PEB at 000000e8a8c9c000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00007ff62fc70000
    NtGlobalFlag:             4400
    NtGlobalFlag2:            0
    Ldr                       00007ffa0ecc53c0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 000002021cc04dc0 . 000002021cc15f00
    Ldr.InLoadOrderModuleList:           000002021cc04f30 . 000002021cc15ee0
    Ldr.InMemoryOrderModuleList:         000002021cc04f40 . 000002021cc15ef0
                Base TimeStamp                     Module
    7ff62fc70000 78facb67 Apr 27 01:06:31 2034 C:\WINDOWS\system32\csrss.exe
    7ffa0eb60000 a52b7c6a Oct 23 22:22:18 2057 C:\WINDOWS\SYSTEM32\ntdll.dll
    7ffa0ba10000 802fce16 Feb 24 11:29:58 2038 C:\WINDOWS\SYSTEM32\CSRSRV.dll
    7ffa0b9f0000 94c740f0 Feb 04 23:17:36 2049 C:\WINDOWS\system32\basesrv.D\
LL

(...)
```

Правильный контекст процесса задается метакомандой `.process`, после чего выводится содержимое `PEB`. Это основной прием для вывода информации, находящейся в пользовательском режиме.

✦ Повторите команду `!process`, но на этот раз без указания уровня детализации. Команда выводит более подробную информацию о процессе:

```
kd> !process ffff8d0e849df080
PROCESS ffff8d0e849df080
    SessionId: 1  Cid: 045c    Peb: e8a8c9c000  ParentCid: 0438
    DirBase: 17afc1002  ObjectTable: ffffe207186d93c0 HandleCount: 1133.
    Image: csrss.exe
    VadRoot ffff8d0e999a4840 Vads 243 Clone 0 Private 672. Modified 48279.
                                  Locked 34\
442.
    DeviceMap ffffe20712213720
    Token                             ffffe207186f38f0
    ElapsedTime                       12:23:30.102
    UserTime                          00:00:00.000
    KernelTime                        00:00:03.468
    QuotaPoolUsage[PagedPool]         422008
    QuotaPoolUsage[NonPagedPool]      37616
```

```
    Working Set Sizes (now,min,max) (1534, 50, 345) (6136KB, 200KB, 1380KB)
    PeakWorkingSetSize                 10222
    VirtualSize                        2101434 Mb
    PeakVirtualSize                    2101467 Mb
    PageFaultCount                     841729
    MemoryPriority                     BACKGROUND
    BasePriority                       13
    CommitCharge                       1014
    Job                                ffff8d0e83da8080
        THREAD ffff8d0e849e0080 Cid 045c.046c Teb: 000000e8a8ca3000
                                                  Win32Thread: f\
fff8d0e865f37c0 WAIT: (WrLpcReceive) UserMode Non-Alertable
            ffff8d0e849e06d8 Semaphore Limit 0x1
        Not impersonating
            DeviceMap ffffe20712213720
            Owning Process ffff8d0e849df080 Image: csrss.exe
            Attached Process N/A Image: N/A
            Wait Start TickCount 2856062 Ticks: 70 (0:00:00:01.093)
            Context Switch Count 6483 IdealProcessor: 8
            UserTime 00:00:00.421
            KernelTime 00:00:00.437
            Win32 Start Address 0x00007ffa0ba15670
            Stack Init ffff83858295fb90 Current ffff83858295f340
            Base ffff838582960000 Limit ffff838582959000 Call 0000000000000000
            Priority 14 BasePriority 13 PriorityDecrement 0
                        IoPriority 2 PagePriority 5
    GetContextState failed, 0x80004001
    Unable to get current machine context, HRESULT 0x80004001
            Child-SP RetAddr Call Site
            ffff8385`8295f380 fffff806`466e98c2 nt!KiSwapContext+0x76
            ffff8385`8295f4c0 fffff806`466e8f54 nt!KiSwapThread+0x3f2
            ffff8385`8295f560 fffff806`466e86f5 nt!KiCommitThreadWait+0x144
            ffff8385`8295f600 fffff806`467d8c56 nt!KeWaitForSingleObject+0x255
            ffff8385`8295f6e0 fffff806`46d76c70 nt!AlpcpWaitForSingleObject+0x3e
            ffff8385`8295f720 fffff806`46d162cc nt!AlpcpCompleteDeferSignal/
RequestAndWait+0x3c
            ffff8385`8295f760 fffff806`46d15321 nt!AlpcpReceiveMessagePort+0x3ac
            ffff8385`8295f7f0 fffff806`46d14e05 nt!AlpcpReceiveMessage+0x361
ffff8385`8295f8d0 fffff806`46885e95 nt!NtAlpcSendWaitReceivePort+0x105
            ffff8385`8295f990 00007ffa`0ebfd194 nt!KiSystemServiceCopyEnd+0x25\
(TrapFrame @ ffff8385`8295fa00)
            000000e8`a8e3f798 00007ffa`0ba15778 0x00007ffa`0ebfd194
            000000e8`a8e3f7a0 00000202`1cc85090 0x00007ffa`0ba15778
            000000e8`a8e3f7a8 00000000`00000000 0x00000202`1cc85090

            THREAD ffff8d0e84bbf140 Cid 045c.066c Teb: 000000e8a8ca9000\
Win32Thread: ffff8d0e865f4760 WAIT: (WrLpcReply) UserMode Non-Alertable\
             ffff8d0e84bbf798 Semaphore Limit 0x1
(...)
```

Команда выводит список всех потоков в процессе. Каждый поток представлен своим адресом ETHREAD, присоединенным к тексту «THREAD». Также указан

стек вызовов; префикс модуля `nt` представляет ядро — нет необходимости использовать «реальное» имя модуля режима ядра.

> Одна из причин для использования `nt` вместо явного указания имени модуля ядра заключается в том, что эти имена различаются в 64- и 32-разрядных системах (ntoskrnl.exe для 64-разрядных систем, как минимум две разновидности для 32-разрядных). Кроме того, такая запись короче.

✦ Символические имена пользовательского режима не загружаются по умолчанию, поэтому в стеках потоков в пользовательском режиме выводятся только числовые адреса. Символические имена можно загрузить явно командой `.reload /user`:

```
lkd> .reload /user
Loading User Symbols
....................
lkd> !process ffff8d0e849df080
PROCESS ffff8d0e849df080
    SessionId: 1  Cid: 045c    Peb: e8a8c9c000  ParentCid: 0438
    DirBase: 17afc1002  ObjectTable: ffffe207186d93c0 HandleCount: 1149.
    Image: csrss.exe

(...)
        THREAD ffff8d0e849e0080 Cid 045c.046c Teb: 000000e8a8ca3000
                        Win32Thread: f\
fff8d0e865f37c0 WAIT: (WrLpcReceive) UserMode Non-Alertable
            ffff8d0e849e06d8 Semaphore Limit 0x1
        Not impersonating
        DeviceMap                 ffffe20712213720
        Owning Process            ffff8d0e849df080 Image: csrss.exe
        Attached Process          N/A Image: N/A
        Wait Start TickCount      2895071 Ticks: 135 (0:00:00:02.109)
        Context Switch Count      6684 IdealProcessor: 8
        UserTime                  00:00:00.437
        KernelTime                00:00:00.437
        Win32 Start Address CSRSRV!CsrApiRequestThread (0x00007ffa0ba15670)
        Stack Init ffff83858295fb90 Current ffff83858295f340
        Base ffff838582960000 Limit ffff838582959000 Call 0000000000000000
        Priority 14 BasePriority 13 PriorityDecrement 0 IoPriority 2 PagePriority 5
GetContextState failed, 0x80004001
Unable to get current machine context, HRESULT 0x80004001
        Child-SP          RetAddr           Call Site
        ffff8385`8295f380 fffff806`466e98c2 nt!KiSwapContext+0x76
        ffff8385`8295f4c0 fffff806`466e8f54 nt!KiSwapThread+0x3f2
        ffff8385`8295f560 fffff806`466e86f5 nt!KiCommitThreadWait+0x144
        ffff8385`8295f600 fffff806`467d8c56 nt!KeWaitForSingleObject+0x255
        ffff8385`8295f6e0 fffff806`46d76c70 nt!AlpcpWaitForSingleObject+0x3e
        ffff8385`8295f720 fffff806`46d162cc nt!AlpcpCompleteDeferSignalRequest/
AndWait+0x3c
```

```
        ffff8385`8295f760 fffff806`46d15321 nt!AlpcpReceiveMessagePort+0x3ac
        ffff8385`8295f7f0 fffff806`46d14e05 nt!AlpcpReceiveMessage+0x361
        ffff8385`8295f8d0 fffff806`46885e95 nt!NtAlpcSendWaitReceivePort+0x105
        ffff8385`8295f990 00007ffa`0ebfd194 nt!KiSystemServiceCopyEnd+0x25\
(TrapFram\e @ ffff8385`8295fa00)
        000000e8`a8e3f798 00007ffa`0ba15778 ntdll!NtAlpcSendWaitReceivePort+0x14
        000000e8`a8e3f7a0 00007ffa`0ebcce7f CSRSRV!CsrApiRequestThread+0x108
        000000e8`a8e3fc30 00000000`00000000 ntdll!RtlUserThreadStart+0x2f

(...)
```

Информацию потока можно просмотреть отдельно командой `!thread` с указанием адреса потока. За описанием различных видов информации, выводимых этой командой, обращайтесь к документации отладчика.

Другие полезные/интересные команды, используемые в отладке режима ядра:

- `!pcr` — вывод структуры PCR (Process Control Region) для процессора, заданного дополнительным индексом (если индекс не указан, по умолчанию отображаются данные для процессора 0).
- `!vm` — вывод статистики использования памяти для систем и процессов.
- `!running` — вывод информации о потоках, работающих на всех процессорах в системе.

Некоторые специализированные команды, используемые при отладке драйверов, будут рассмотрены в следующих главах.

Полная отладка режима ядра

Полная отладка режима ядра требует предварительной настройки на хосте и управляемой машине. В этом разделе я покажу, как настроить виртуальную машину в качестве управляемой для отладки режима ядра. Это рекомендуемая и самая удобная конфигурация для работы над драйверами режима ядра (но не для разработки драйверов устройств для оборудования). В этом разделе рассматриваются основные этапы настройки виртуальной машины (VM) Hyper-V. Если вы используете другую технологию виртуализации (например, VMWare или VirtualBox), обращайтесь к документации фирмы-разработчика или к ресурсам в интернете за описанием процесса настройки.

Управляемая машина должна взаимодействовать с хостом по некоторому каналу связи. Есть несколько вариантов, самый лучший — использование сети. К сожалению, для этого хост и управляемая машина должны работать под управлением как минимум Windows 8. Так как Windows 7 все еще остается до-

пустимой целью управления, мы воспользуемся другим вариантом — последовательным портом (COM). Конечно, на многих современных компьютерах уже нет последовательных портов, а при подключении к VM физические кабели не нужны. Все платформы виртуализации позволяют перенаправить виртуальный последовательный порт в именованный канал на хосте; именно эта конфигурация будет использована в нашем примере.

Настройка управляемой машины

Управляемая VM должна быть настроена для отладки режима ядра по аналогии с локальной отладкой режима ядра, но с дополнительным каналом связи, подключенным к виртуальному последовательному порту на этой машине.

Один из способов настройки основан на выполнении `bcdedit` в окне командной строки с повышенными привилегиями:

```
bcdedit /debug on
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Задайте номер порта отладки в соответствии с фактическим номером виртуального последовательного порта (обычно 1).

Чтобы изменения вступили в силу, VM необходимо перезапустить. Но перед этим можно связать последовательный порт с именованным каналом. Для виртуальных машин Hyper-V процедура выглядит так:

- Если Hyper-V VM относится к поколению 1 (более старому), в настройках VM имеется простой пользовательский интерфейс для изменения конфигурации. Добавьте последовательный порт командой Add Hardware, если ни один порт не был определен ранее. Затем настройте последовательный порт и свяжите его с именованным портом по вашему выбору. Диалоговое окно настройки изображено на рис. 5.6.
- Для VM поколения 2 пользовательский интерфейс не нужен. Чтобы выполнить настройку, убедитесь в том, что VM закрыта (хотя это не обязательно в самых последних версиях Windows 10), и откройте окно PowerShell с повышенными привилегиями.
- Введите следующую команду, чтобы связать последовательный порт с именованным каналом:

```
Set-VMComPort myvmname -Number 1 -Path \\.\pipe\debug
```

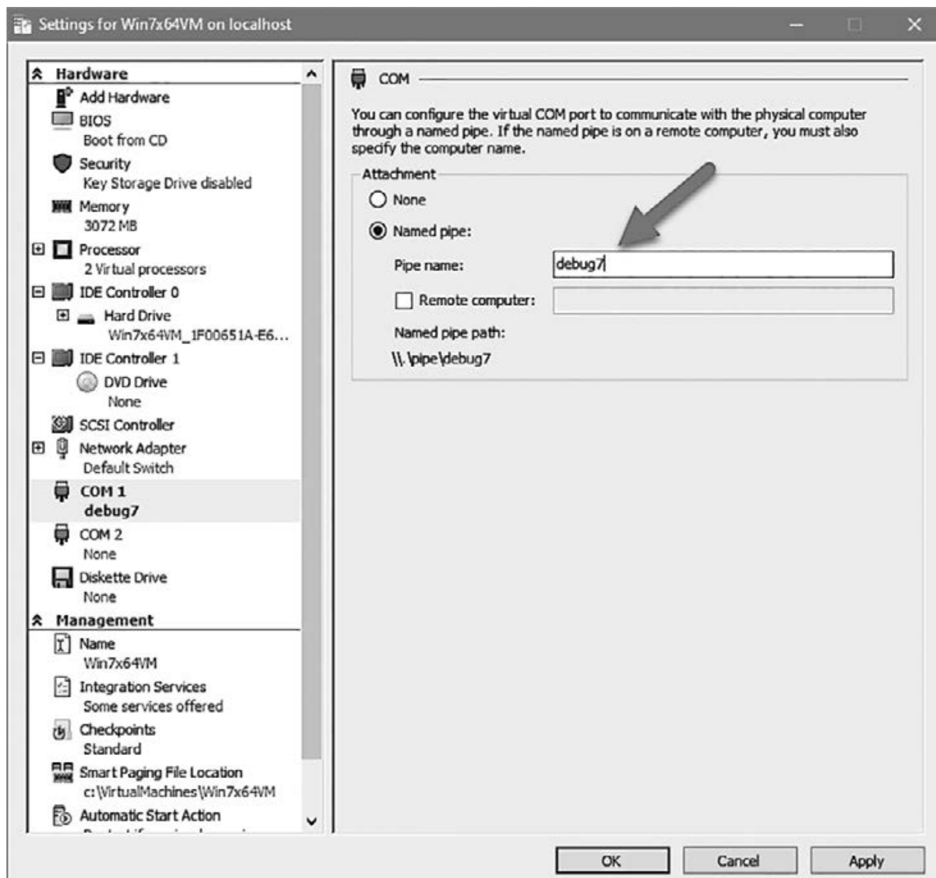


Рис. 5.6. Отображение последовательных портов на именованные каналы для Hyper-V Gen-1 VM

Измените имя VM соответствующим образом и задайте номер COM-порта из приведенной выше команды `bcdedit`. Проследите за тем, чтобы путь к каналу был уникальным.

✦ Убедитесь в том, что настройки имеют ожидаемые значения, при помощи команды `Get-VMComPort`:

```
Get-VMComPort myvmname

VMName Name Path
------ ---- ----
myvmname COM 1 \\.\pipe\debug
myvmname COM 2
```

Загрузите VM — управляемая машина готова.

Настройка хоста

Отладчик режима ядра должен быть настроен для подключения к VM на том же последовательном порте, который был связан с именованным каналом, предоставляемым хостом.

- Запустите отладчик режима ядра и выберите команду File ▸ Attach To Kernel. Перейдите на вкладку COM. Заполните необходимые подробности, настроенные на управляемой машине. На рис. 5.7 показано, как выглядят эти настройки.

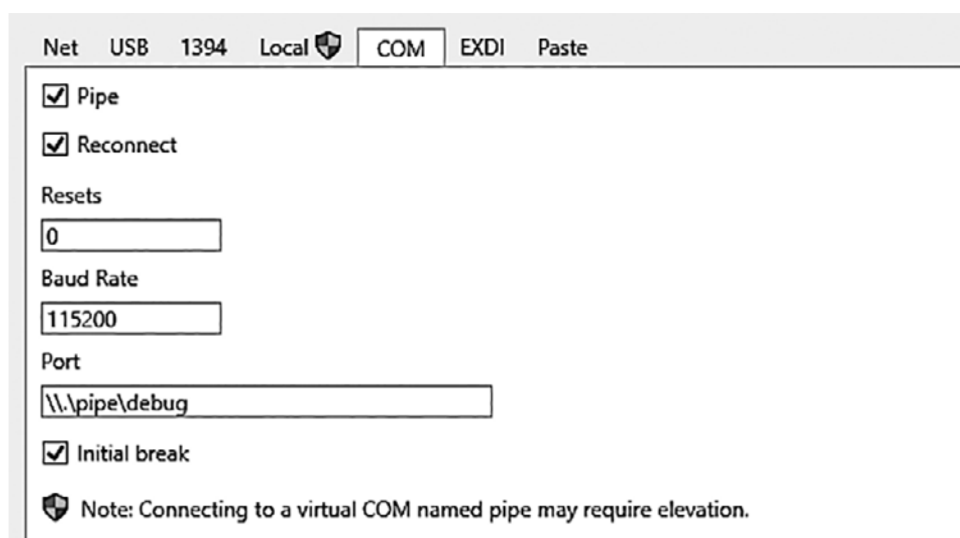


Рис. 5.7. Настройка конфигурации COM-порта на хосте

Щелкните на кнопке OK. Отладчик должен присоединиться к управляемой машине. Если этого не произойдет, щелкните на кнопке Break на панели инструментов.

Типичный результат выглядит так:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\debug
Waiting to reconnect...
Connected to Windows 10 18362 x64 target at (Sun Apr 21 11:28:11.300 2019
                                            (UTC + 3:0\
0)), ptr64 TRUE
Kernel Debugger connection established. (Initial Breakpoint requested)
```

```
************* Path validation summary **************
Response                         Time (ms)      Location
Deferred                                        SRV*c:\Symbols*http://msdl.
microsof
com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff801`36a09000 PsLoadedModuleList = 0xfffff801`36e4c2d0
Debug session time: Sun Apr 21 11:28:09.669 2019 (UTC + 3:00)
System Uptime: 1 days 0:12:28.864
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
* You are seeing this message because you pressed either                      *
* CTRL+C (if you run console kernel debugger) or,                             *
* CTRL+BREAK (if you run GUI kernel debugger),                                *
* on your debugger machine's keyboard.                                        *
*                                                                             *
* THIS IS NOT A BUG OR A SYSTEM CRASH                                         *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now. This message might immediately reappear. If it   *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!DbgBreakPointWithStatus:
fffff801`36bcd580 cc              int 3
```

Обратите внимание: в приглашении появляется индекс и слово kd. Индекс относится к текущему процессору, который вызвал прерывание. В этот момент управляемая VM полностью заблокирована. Теперь можно выполнять отладку обычным образом, помня о том, что при каждом прерывании вся машина приостанавливается.

Основы отладки режима ядра

После того как хост будет соединен с управляемой машиной, можно приступить к отладке. Для демонстрации полной отладки режима ядра будет использоваться драйвер PriorityBooster, который был создан в главе 4.

- Установите (но не загружайте) драйвер на управляемой машине, как это было сделано в главе 4. Убедитесь в том, что вместе с SYS-файлом самого

драйвера был скопирован PDB-файл. Это упростит получение правильных символических имен для драйвера.

- Установите точку прерывания в функции `DriverEntry`. Загрузить драйвер нельзя, потому что это приведет к выполнению `DriverEntry`, и установить точку прерывания в этой функции уже не удастся. Так как драйвер еще не загружен, можно воспользоваться командой `bu` (Unresolved Breakpoint) для установки будущей точки прерывания. Переключитесь на управляемую машину, если она выполняется в данный момент, и введите следующую команду:

```
0: kd> bu prioritybooster!driverentry
0: kd> bl
   0 e Disable Clear u        0001 (0001) (prioritybooster!driverentry)
```

Точка прерывания не отрабатывает, так как модуль еще не загружен.

Введите команду `g`, чтобы разрешить управляемой машине продолжить работу, и загрузите драйвер командой `sc start booster` (в предположении, что драйверу было сохранено имя booster). Если все прошло нормально, точка прерывания должна сработать, и исходный файл должен загрузиться автоматически, как показывает следующий вывод в окне командной строки:

```
0: kd> g
Breakpoint 0 hit
PriorityBooster!DriverEntry:
fffff801`358211d0 4889542410       mov qword ptr [rsp+10h],rdx
```

На рис. 5.8 изображен снимок экрана: окно исходного кода WinDbg Preview автоматически открывается с выделением правильной строки. Также отображается окно Locals, как и следовало ожидать.

На этой стадии вы можете выполнять строки исходного кода в пошаговом режиме, просматривать переменные в окне Locals и даже добавлять выражения в окне Watch. Также возможно изменять значения в окне Locals, как и во многих других отладчиках.

Окно командной строки остается доступным, как обычно, но некоторые операции проще выполнять в пользовательском интерфейсе. Например, точки прерывания можно устанавливать обычной командой `bp`, но также можно открыть файл с исходным кодом (если он не был открыт ранее), перейти к строке, в которой устанавливается точка прерывания, и нажать F9 или щелкнуть на соответствующей кнопке на панели инструментов. В любом случае в окне командной строки будет выполнена команда `bp`. Окно Breakpoints содержит краткий список точек прерывания, установленных на текущий момент.

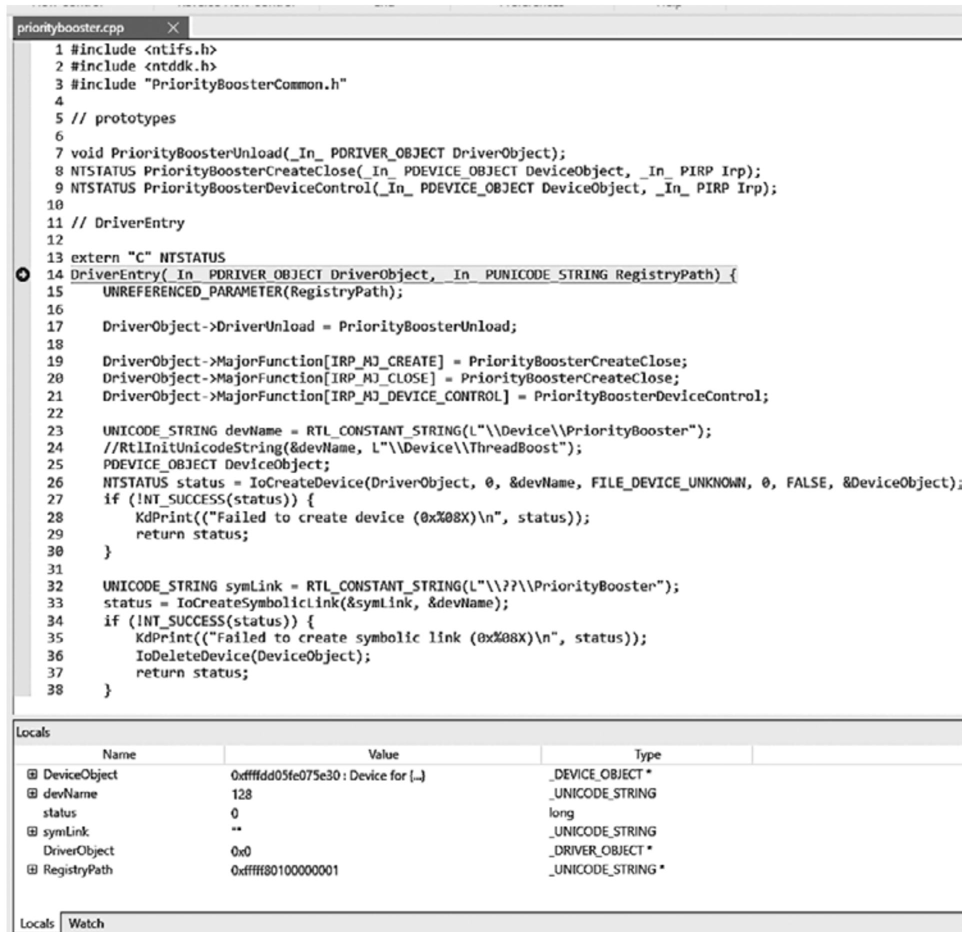


Рис. 5.8. Сработавшая точка прерывания в DriverEntry

✦ Введите команду `k`, чтобы увидеть, как вызывается `DriverEntry`:

```
2: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffad08`226df898 fffff801`35825020 PriorityBooster!DriverEntry
                                       [c:\dev\priorityb\
ooster\prioritybooster\prioritybooster.cpp @ 14]
01 ffffad08`226df8a0 fffff801`37111436 PriorityBooster!GsDriverEntry+0x20
                                       [minkernel\
\tools\gs_support\kmodefastfail\gs_driverentry.c @ 47]
02 ffffad08`226df8d0 fffff801`37110e6e nt!IopLoadDriver+0x4c2
03 ffffad08`226dfab0 fffff801`36ab7835 nt!IopLoadUnloadDriver+0x4e
04 ffffad08`226dfaf0 fffff801`36b39925 nt!ExpWorkerThread+0x105
```

```
05 ffffad08`226dfb90 fffff801`36bccd5a nt!PspSystemThreadStartup+0x55
06 ffffad08`226dfbe0 00000000`00000000 nt!KiStartSystemThread+0x2a
```

Если точки прерывания не устанавливаются, возможно, проблема с символическими именами. Выполните команду `.reload` и посмотрите, не исчезла ли проблема. Установка точек прерывания в пользовательском режиме возможна, но сначала нужно выполнить команду `.reload /user`, которая поможет вам в этом.

Возможно, точка прерывания должна срабатывать только в том случае, когда код выполняется в конкретном процессе. Это можно сделать добавлением ключа `/p` к точке прерывания. В следующем примере точка прерывания устанавливается только в том случае, если процессом является process.exe:

```
2: kd> !process 0 0 explorer.exe
PROCESS ffffdd06042e4080
    SessionId: 2 Cid: 1df8 Peb: 00dee000 ParentCid: 1dd8
    DirBase: 1bf58a002 ObjectTable: ffff960a682133c0 HandleCount: 3504.
    Image: explorer.exe

2: kd> bp /p ffffdd06042e4080 prioritybooster!prioritybooster devicecontrol
2: kd> bl
     0 e Disable Clear fffff801`358211d0 [c:\dev\prioritybooster\
prioritybooster\p\
riorityboster.cpp @ 14] 0001 (0001) PriorityBooster!DriverEntry
     1 e Disable Clear fffff801`35821040 [c:\dev\prioritybooster\
prioritybooster\p\
riorityboster.cpp @ 63] 0001 (0001) PriorityBooster!PriorityBoosterDevice\
Control
     Match process data ffffdd06`042e4080
```

Установите обычную точку прерывания в `switch case` для кода управления вводом/выводом; нажмите F9 на строке в окне исходного кода, как показано на рис. 5.9 (и удалите условную точку прерывания, нажав F9 на этой строке).

```
62 _Use_decl_annotations_
63 NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
64     // get our IO_STACK_LOCATION
65     auto stack = IoGetCurrentIrpStackLocation(Irp);
66     auto status = STATUS_SUCCESS;
67
68     switch (stack->Parameters.DeviceIoControl.IoControlCode) {
69         case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
70         {
71             // do the work
72             if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(ThreadData))
73                 status = STATUS_BUFFER_TOO_SMALL;
74                 break;
75             }
```

Рис. 5.9. Точка прерывания в DriverEntry

- Запустите тестовое приложение с указанием идентификатора процесса и приоритета:

```
booster 2000 30
```

Точка прерывания должна сработать. Вы можете продолжить нормальную отладку, сочетая работу на уровне исходного кода с вводом команд.

Итоги

В этой главе были рассмотрены основы отладки с использованием WinDbg. Навыки отладки исключительно полезны для разработчика, так как программы любых видов, включая драйверы ядра, могут содержать ошибки.

В следующей главе описаны некоторые механизмы режима ядра, которые необходимо знать каждому разработчику, потому что они часто встречаются при разработке и отладке драйверов.

Глава 6

Механизмы режима ядра

В этой главе рассматриваются различные механизмы, предоставляемые ядром Windows. Некоторые из них могут напрямую использоваться разработчиками драйверов. Другие механизмы разработчик драйвера должен хотя бы понимать, так как они принесут пользу при отладке и позволять понять происходящее в системе.

В этой главе:

- Уровень запроса прерывания
- Отложенные вызовы процедур
- Асинхронные вызовы процедур
- Структурированная обработка исключений
- Фатальный сбой системы
- Синхронизация потоков
- Высокоуровневая синхронизация IRQL
- Рабочие элементы

Уровень запроса прерывания

В главе 1 рассматривались потоки и приоритеты потоков. Приоритеты учитываются в ситуациях, в которых количество выполненных потоков превышает количество доступных процессоров. В то же время физические устройства должны уведомлять систему о том, что нечто требует ее внимания. Простейший пример — операция ввода/вывода, выполняемая дисковым накопителем. После завершения операции устройство должно уведомить систему о завершении, выдав запрос на *прерывание*. Прерывание связано со специальным устройством — контроллером прерываний, который затем отправляет запрос

процессору для обработки. Возникает следующий вопрос: в каком потоке должен выполняться соответствующий обработчик прерывания (ISR, Interrupt Service Routine)?

С каждым аппаратным прерыванием связан специальный приоритет, который определяется уровнем HAL — он называется уровнем запроса прерывания, или IRQL (Interrupt Request Level) (не путайте с физической линией прерывания IRQ!). Контекст каждого процессора имеет собственное значение IRQL наряду с обычными регистрами. IRQL может быть или не быть реализован оборудованием процессора — по сути, это неважно. IRQL следует рассматривать как любой другой регистр процессора.

Основное правило гласит, что процессор выполняет код с наивысшим значением IRQL. Например, если значение IRQL процессора в некоторый момент равно 0 и поступает прерывание, с которым связано значение IRQL 5, процессор сохраняет свое состояние (контекст) в стеке режима ядра текущего потока, повышает свое значение IRQL до 5, после чего выполняется обработчик, связанный с прерыванием. После завершения обработчика IRQL падает до предыдущего уровня, а ранее выполнявшийся код возобновляет выполнение так, словно никакого прерывания не было. Во время выполнения обработчика другие прерывания, поступающие с уровнем IRQL 5 и ниже, не прервут работу этого процессора. С другой стороны, если IRQL нового прерывания выше 5, процессор снова сохранит свое состояние, поднимет IRQL до нового уровня и выполнит второй обработчик, связанный со вторым прерыванием. При его завершении уровень IRQL снова упадет до 5, процессор восстановит свое состояние и продолжит выполнение исходного обработчика. Фактически повышение IRQL временно блокирует выполнение кода с равным либо меньшим уровнем IRQL. Основная последовательность событий при возникновении прерывания изображена на рис. 6.1. На рис. 6.2 показано, как выглядит вложение прерываний.

У сценариев на рис. 6.1 и 6.2 есть одна важная особенность: все обработчики выполняются в том потоке, который был изначально прерван. В Windows нет специального потока для обработки прерываний; они обрабатываются потоком, который выполнялся в этот момент на прерываемом процессоре. Как вскоре будет показано, переключение контекста невозможно, если IRQL процессора равен 2 или выше, поэтому во время выполнения этих обработчиков поток не может тайно проникнуть на процессор.

> Квант прерванного потока не уменьшается из-за этих «прерываний» — условно говоря, он в этом не виноват.

Рис. 6.1. Основной механизм диспетчеризации прерываний

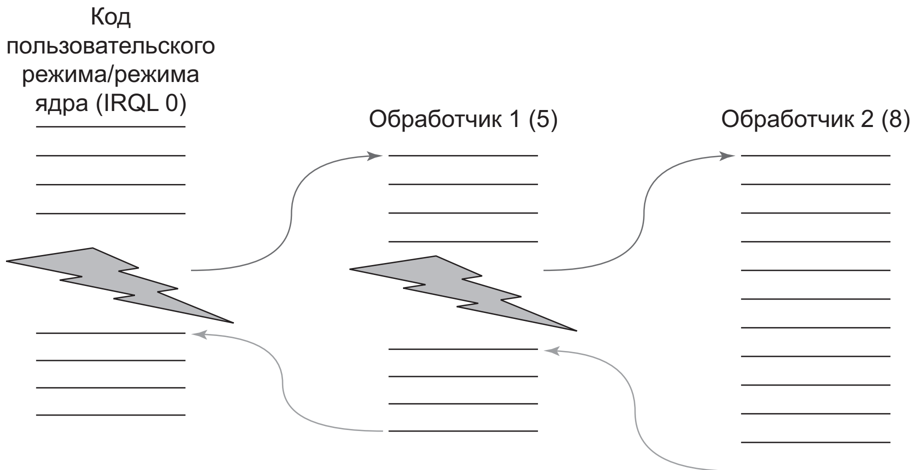


Рис. 6.2. Вложенные прерывания

При выполнении кода пользовательского режима значение IRQL всегда равно 0. Это одна из причин, по которым термин IRQL не встречается в документации пользовательского режима — значение всегда равно 0 и не может быть изменено. Большая часть кода режима ядра также выполняется с нулевым IRQL. В режиме ядра возможно поднять значение IRQL для текущего процессора.

Важнейшие значения IRQL описаны ниже:

- `PASSIVE_LEVEL` в WDK (`0`) — «нормальный» уровень IRQL для процессора. Код пользовательского режима всегда выполняется на этом уровне. Планирование потоков работает как обычно (см. главу 1).
- `APC_LEVEL` (`1`) — используется для специальных APC-вызовов режима ядра (асинхронный вызов процедур будет рассмотрен позднее в этой главе). Планирование потоков работает нормально.
- `DISPATCH_LEVEL` (`2`) — на этом уровне все радикально меняется. Планировщик не может активизироваться на этом процессоре. Обращения к выгруженной памяти недопустимы — попытки приводят к сбою системы. Так как планировщик выполняться не может, ожидание по объектам режима ядра недопустимо (при попытке использования происходит фатальный сбой системы).
- IRQL устройств — диапазон уровней, используемых для аппаратных прерываний (от 3 до 11 для x64/ARM/ARM64, от 3 до 26 для x86). Здесь действуют правила, относящиеся к IRQL 2.
- Наивысший уровень (`HIGH_LEVEL`) — самый высокий уровень IRQL, блокирующий все прерывания. Используется некоторыми функциями API, предназначенными для манипуляций со связными списками. Фактические значения — 15 (x64/ARM/ARM64) и 31 (x86).

Когда уровень IRQL процессора поднимается до 2 и выше (по любой причине), для выполняемого кода устанавливаются определенные ограничения:

- Обращение к адресам, не находящимся в физической памяти, приводит к фатальному сбою системы. Это означает, что обращение к данным из невыгружаемого пула всегда безопасно, тогда как обращение к данным из выгружаемого пула или из буферов, предоставляемых пользователем, небезопасно, и его следует избегать.
- Ожидание по любому объекту диспетчеризации режима ядра (например, мьютексу или событию) приводит к фатальному сбою системы, если только тайм-аут ожидания не равен 0, что разрешено (объекты диспетчеризации и ожидание будут рассматриваться позднее в разделе «Синхронизация» этой главы).

Эти ограничения обусловлены тем фактом, что планировщик «выполняется» на уровне IRQL 2; таким образом, если IRQL процессора уже находится на уровне 2 и выше, планировщик не может активизироваться на этом процессоре, поэтому переключение контекста (заменяющее выполняемый поток другим потоком на процессоре) произойти не может. Только прерывания более высокого уровня могут временно отклонить выполнение кода на соответствующий обработчик прерывания, но поток остается тем же. Контекст потока сохраняется, обработчик переключения выполняется, и состояние потока восстанавливается.

В процессе отладки значение IRQL текущего процессора можно просмотреть командой `!irql`. Также можно дополнительно указать номер процессора, чтобы просмотреть IRQL заданного процессора.

Для просмотра зарегистрированных прерываний в системе можно воспользоваться командой отладчика `!idt`.

Повышение и понижение IRQL

Как упоминалось ранее, в пользовательском режиме концепция IRQL не упоминается, и изменить значение IRQL не удастся. В режиме ядра функция `KeRaiseIrql` повышает уровень IRQL, а функция `KeLowerIrql` — понижает его. Следующий фрагмент кода повышает IRQL до `DISPATCH_LEVEL` (2), после чего снова понижает его после выполнения команд с новым IRQL.

```
// Предполагается, что текущее значение IRQL <= DISPATCH_LEVEL

KIRQL oldIrql; // Определяется как UCHAR в typedef
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);

NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);

// Выполнить работу на уровне IRQL DISPATCH_LEVEL

KeLowerIrql(oldIrql);
```

Поднимая уровень IRQL, позаботьтесь о том, чтобы он был понижен в той же функции. Возвращаться из функции с IRQL выше того, с которым вы вошли в нее, слишком опасно. Кроме того, убедитесь в том, что `KeRaiseIrql` действительно повышает уровень IRQL, а `KeLowerIrql` действительно понижает его; в противном случае произойдет фатальный сбой системы.

Приоритеты потоков и IRQL

IRQL является атрибутом процессора, а приоритет является атрибутом потока. Приоритеты потоков имеют смысл только при IRQL<2. Как только выполняющийся поток поднимет IRQL до 2 и выше, его приоритет перестает что-либо означать (теоретически он имеет бесконечный квант времени), и он продолжает выполняться до тех пор, пока не понизит IRQL до значения меньше 2.

Естественно, проводить много времени при IRQL >= 2 нежелательно; это гарантированно исключает выполнение кода пользовательского режима. Это всего лишь одна из причин, по которым для того, что выполняемый код может делать на этих уровнях, устанавливаются жесткие ограничения.

В Диспетчере задач время, проводимое процессором на IRQL 2 и выше, отображается в форме псевдопроцесса с именем `System Interrupts`; в Process Explorer он называется `Interrupts`. На рис. 6.3 показан снимок экрана из Диспетчера задач, а на рис. 6.4 изображена та же информация в Process Explorer.

Processes | Performance | App history | Startup | Users | Details | Services

Name	PID	Status	User name	Ses...	CPU	Memory (a...	Commit size	Base priority	H...	Th...	Description
System interrupts	-	Running	SYSTEM	0	01	0 K	0 K	N/A	-	-	Deferred procedure calls and interrupt service routines
System Idle Process	0	Running	SYSTEM	0	86	8 K	60 K	N/A	-	12	Percentage of time the processor is idle
System	4	Running	SYSTEM	0	00	20 K	204 K	N/A	9,...	382	NT Kernel & System

Рис. 6.3. Время процессора на IRQL 2+ в Диспетчере задач

Process	PID	CPU	Private Bytes	Working Set	Description	User Name
Interrupts	n/a	1.06	0 K	0 K	Hardware Interrupts and DPCs	
System Idle Process	0	83.28	60 K	8 K		NT AUTHORITY\SYSTEM
System	4	0.88	204 K	3,932 K		NT AUTHORITY\SYSTEM
Secure System	88	Suspended	184 K	80,372 K		NT AUTHORITY\SYSTEM

Рис. 6.4. Время процессора на IRQL 2+ в Process Explorer

Отложенные вызовы процедур

На рис. 6.5 изображена типичная последовательность событий при выполнении клиентом некоторой операции ввода/вывода. На рисунке поток пользовательского режима открывает дескриптор файла и инициирует операцию чтения функцией `ReadFile`. Так как поток может выдать асинхронный вызов, он почти немедленно восстанавливает управление и может заняться другой работой. Драйвер, получивший запрос, вызывает драйвер файловой системы (например, NTFS), который может обратиться с вызовом к драйверам более низкого уровня, пока запрос не достигнет драйвера диска, инициирующего операцию с реальным оборудованием диска. При этом никакому коду выполняться не нужно, а оборудование просто «делает свое дело».

Когда оборудование завершает операцию чтения, оно выдает прерывание. Это приводит к выполнению обработчика, связанного с этим прерыванием, с уровнем IRQL устройства (следует заметить, что поток, обрабатывающий запрос, выбирается произвольно, так как прерывание поступает асинхронно). Типичный обработчик обращается к оборудованию устройства для получения результата операции. Его последним шагом становится завершение исходного запроса.

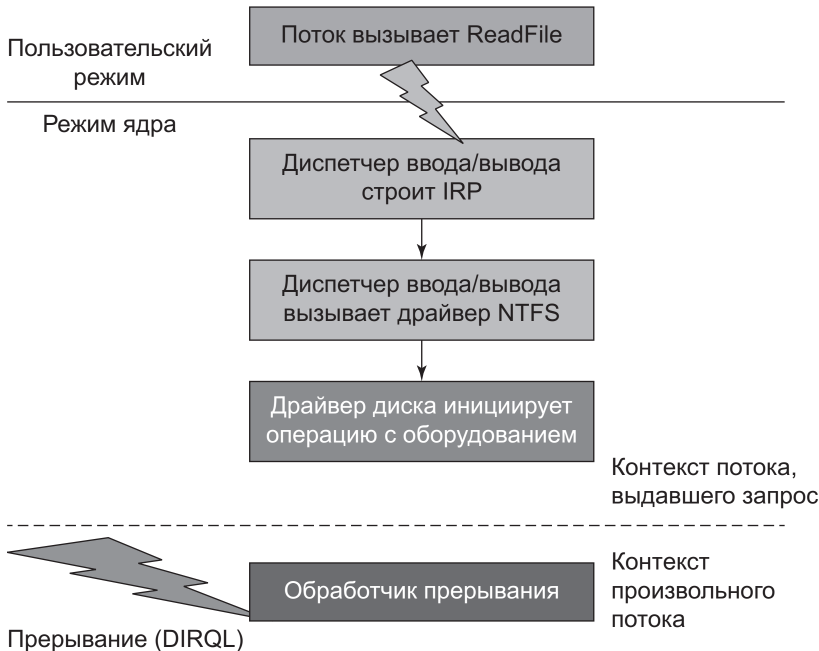


Рис. 6.5. Обработка типичного запроса ввода/вывода (часть 1)

Как было показано в главе 4, завершение запроса осуществляется вызовом `IoCompleteRequest`. Однако в документации указано, что эта функция может вызываться только с IRQL<=`DISPATCH_LEVEL`(2). А следовательно, обработчик не может вызвать `IoCompleteRequest`, поскольку это приведет к фатальному сбою системы. Что же делать обработчику?

«Почему существует такое ограничение?» — спросите вы. Одна из причин связана с тем, что делает `IoCompleteRequest`. Эта тема будет более подробно рассмотрена в следующей главе, а пока в нескольких словах скажу, что вызов функции обходится относительно дорого. Если бы вызов был разрешен, это означало бы, что выполнение обработчика занимало гораздо больше времени. Выполнение на высоком уровне IRQL блокирует другие прерывания на более длительный период времени.

Механизм, который позволяет обработчику вызвать IoCompleteRequest (и другие функции с похожими ограничениями) при первой возможности, называется *отложенным вызовом процедуры*, или DPC (Deferred Procedure Call). DPC-вызов — объект, инкапсулирующий функцию, которая должна вызываться на уровне IRQL DISPATCH_LEVEL. На этом уровне IRQL вызов IoCompleteRequest разрешен.

Почему бы обработчику просто не понизить текущий уровень IRQL до DISPATCH_LEVEL, вызвать IoCompleteRequest, а затем снова повысить IRQL до исходного значения? Дело в том, что это может привести к взаимной блокировке (deadlock). Причина более подробно объясняется в разделе «Спин-блокировки».

Драйвер, зарегистрировавший обработчик прерывания, подготавливает DPC-вызов заранее; для этого он выделяет память для структуры KDPC в невыгружаемом пуле и инициализирует его функцией обратного вызова с использованием KeInitializeDpc. Затем при вызове обработчика, непосредственно перед выходом из функции, очередь приказывает DPC-вызову выполниться как можно скорее, ставя его в очередь вызовом KeInsertQueueDpc. При выполнении функция DPC вызывает IoCompleteRequest. Таким образом, DPC-вызов служит своего рода компромиссом — код выполняется на уровне IRQL DISPATCH_LEVEL, что означает невозможность планирования, доступа к выгружаемой памяти и т. д., но недостаточно высоком для того, чтобы помешать поступлению и обработке аппаратных прерываний на том же процессоре.

Каждый процессор в системе имеет собственную очередь DPC-вызовов. По умолчанию KeInsertQueueDpc ставит DPC-вызов в очередь DPC текущего процессора. При возврате управления из обработчика, прежде чем IRQL сможет снова упасть до нуля, проверяется наличие DPC-вызовов в очереди процессора. Если DPC-вызовы присутствуют, процессор уменьшает IRQL до уровня DISPATCH_LEVEL (2), после чего обрабатывает DPC-вызовы в очереди по принципу FIFO (First In First Out, то есть «первым зашел, первым вышел») и вызывает соответствующие функции, пока очередь не опустеет. Только после этого уровень IRQL процессора уменьшается до нуля, и продолжается выполнение кода, который был приостановлен в момент поступления прерывания.

У объектов DPC-вызовов можно настраивать некоторые параметры. Обращайтесь к документации функций KeSetImportantceDpc и KeSetTargetProcessorDpc.

На рис. 6.6 схема на рис. 6.5 дополняется выполнением функции DPC.

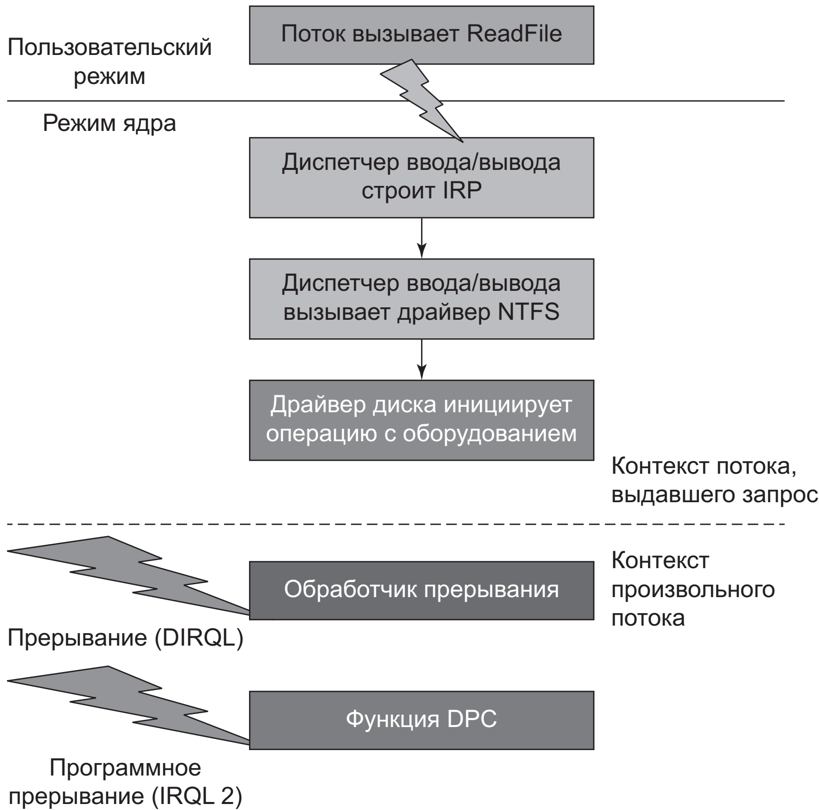


Рис. 6.6. Обработка типичного запроса ввода/вывода (часть 2)

Использование DPC с таймером

Объекты DPC-вызовов изначально создавались для использования с обработчиками прерываний. Тем не менее в ядре существуют и другие механизмы, использующие DPC-вызовы.

В частности, они используются таймером режима ядра. Таймер режима ядра, представленный структурой `KTIMER`, позволяет установить таймер на некоторый момент времени в будущем — по истечении относительного интервала или на абсолютный момент времени. Таймер является объектом диспетчеризации, поэтому он может использоваться для ожидания функцией `KeWaitForSingleObject` (см. раздел «Синхронизация» этой главы). Хотя ожидание возможно, с таймерами оно неудобно. Более простое решение — активизация функции обратного вызова при истечении таймера. Именно такую возможность предоставляет

таймер ядра, при этом объект DPC-вызова используется в качестве функции обратного вызова.

Следующий фрагмент кода показывает, как настроить таймер и связать его с DPC-вызовом. При истечении таймера DPC-вызов ставится в очередь DPC процессора и выполняется при первой возможности. DPC-вызовы мощнее функций обратного вызова на базе нулевого уровня IRQL, потому что DPC-вызов гарантированно будет выполнен ранее любого кода пользовательского режима (и большей части кода режима ядра).

```
KTIMER Timer;
KDPC TimerDpc;

void InitializeAndStartTimer(ULONG msec) {
    KeInitializeTimer(&Timer);
    KeInitializeDpc(&TimerDpc,
        OnTimerExpired, // Функция обратного вызова
        nullptr); // Передается функции обратного вызова как "контекст"

    // Относительный интервал задается в 100-наносекундных единицах
    // (и должен быть отрицательным).
    // Чтобы преобразовать в миллисекунды, умножьте значение на 10000.

    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * msec;
    KeSetTimer(&Timer, interval, &TimerDpc);
}

void OnTimerExpired(KDPC* Dpc, PVOID context, PVOID, PVOID) {
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(context);
    NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);
    // Обработка истечения времени таймера
}
```

Асинхронные вызовы процедур

Как было показано в предыдущем разделе, объект DPC-вызова инкапсулирует функцию, выполняемую на уровне IRQL `DISPATCH_LEVEL`. Для DPC-вызова неважно, от какого именно потока поступил вызов.

Асинхронные вызовы процедур (APC, Asynchronous Procedure Call) также представляют собой структуры данных, инкапсулирующие вызываемую функцию. Но в отличие от DPC-вызов, APC-вызов предназначен для конкретного потока, так что только этот поток сможет выполнить функцию. Следовательно, с каждым потоком связана отдельная очередь APC-вызовов.

Существуют три разновидности APC-вызовов:

- *APC-вызовы пользовательского режима* — выполняются в пользовательском режиме на уровне IRQL `PASSIVE_LEVEL`, только когда поток переходит в *тревожное* (alertable) состояние. Обычно эта задача решается вызовом таких функций API, как `SleepEx`, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` и т. д. Последний аргумент этих функций может быть задан равным `TRUE` для перевода потока в тревожное состояние. В этом состоянии он проверяет состояние очереди APC, и если она не пуста, APC-вызовы выполняются до тех пор, пока очередь не опустеет.
- *Нормальные APC-вызовы режима ядра* — выполняются в режиме ядра на уровне IRQL `PASSIVE_LEVEL` с вытеснением кода пользовательского режима и APC-вызовов пользовательского режима.
- *Специальные APC-вызовы режима ядра* — выполняются в режиме ядра на уровне IRQL `APC_LEVEL` (1) с вытеснением кода пользовательского режима, нормальных APC-вызовов режима ядра и APC-вызовов пользовательского режима. Эти APC-вызовы используются системой ввода/вывода для завершения операций ввода/вывода. Стандартный сценарий такого рода будет рассмотрен в следующей главе.

Функции API для работы с APC-вызовами не документированы в режиме ядра, поэтому драйверы обычно не используют APC-вызовы напрямую.

Пользовательский режим может использовать APC-вызовы (пользовательского режима) при помощи некоторых функций API. Например, вызов `ReadFileEx` или `WriteFileEx` начинает асинхронную операцию ввода/вывода. Когда операция завершится, APC-вызов пользовательского режима присоединяется к вызывающему потоку. Этот APC-вызов выполнится, когда поток войдет в тревожное состояние (см. выше). Другая полезная функция пользовательского режима для явного генерирования APC-вызовов — `QueueUserAPC`. За дополнительной информацией обращайтесь к документации Windows API.

Критические секции и защищенные секции

Критическая секция запрещает выполнение APC-вызовов пользовательского режима и нормальных APC-вызовов режима ядра (специальные APC-вызовы режима ядра все равно могут выполняться). Поток входит в критическую секцию вызовом функции `KeEnterCriticalRegion` и выходит из нее вызовом `KeLeaveCriticalRegion`. Некоторые функции ядра могут вызываться только в критической секции, особенно при работе с ресурсами исполнительной системы (см. раздел «Ресурсы исполнительной системы» далее в этой главе).

Защищенная секция блокирует выполнение любых APC-вызовов. Функция `KeEnterGuardedRegion` вызывается для входа в защищенную секцию, а функция `KeLeaveGuardedRegion` — для выхода из нее. Рекурсивные вызовы `KeEnterGuardedRegion` должны компенсироваться равным количеством вызовов `KeLeaveGuardedRegion`.

Повышение уровня IRQL до APC_LEVEL блокирует доставку всех APC-вызовов.

Структурированная обработка исключений

Исключение (exception) представляет собой событие, которое происходит из-за того, что некоторая команда привела к выдаче ошибки процессором. Исключения в каком-то смысле похожи на прерывания; главное отличие заключается в том, что исключения синхронны и технически воспроизводимы при тех же условиях, тогда как прерывание асинхронно и может поступить в любой момент времени. Примеры исключений — деление на нуль, точки прерываний, ошибки страниц, переполнение стека и недопустимые команды.

При возникновения исключения ядро перехватывает его и разрешает коду обработать это исключение, если это возможно. Этот механизм называется *структурированной обработкой исключений*, или SEH (Structured Exception Handling); он доступен как для кода пользовательского режима, так и для кода режима ядра.

Обработчики исключений режима ядра вызываются по таблице IDT (Interrupt Dispatch Table) — той же таблице, в которой хранятся соответствия между векторами и обработчиками прерываний. Все эти соответствия выводятся командой `!idt` в отладчике ядра. В сущности, векторы прерываний с низкими номерами являются обработчиками исключений. Пример вывода команды:

```
lkd> !idt

Dumping IDT: fffff8011d941000

00: fffff8011dd6c100 nt!KiDivideErrorFaultShadow
01: fffff8011dd6c180 nt!KiDebugTrapOrFaultShadow Stack = 0xFFFFF8011D9459D0
02: fffff8011dd6c200 nt!KiNmiInterruptShadow     Stack = 0xFFFFF8011D9457D0
03: fffff8011dd6c280 nt!KiBreakpointTrapShadow
04: fffff8011dd6c300 nt!KiOverflowTrapShadow
05: fffff8011dd6c380 nt!KiBoundFaultShadow
06: fffff8011dd6c400 nt!KiInvalidOpcodeFaultShadow
```

```
07: fffff8011dd6c480 nt!KiNpxNotAvailableFaultShadow
08: fffff8011dd6c500 nt!KiDoubleFaultAbortShadow Stack = 0xFFFFF8011D9453D0
09: fffff8011dd6c580 nt!KiNpxSegmentOverrunAbortShadow
0a: fffff8011dd6c600 nt!KiInvalidTssFaultShadow
0b: fffff8011dd6c680 nt!KiSegmentNotPresentFaultShadow
0c: fffff8011dd6c700 nt!KiStackFaultShadow
0d: fffff8011dd6c780 nt!KiGeneralProtectionFaultShadow
0e: fffff8011dd6c800 nt!KiPageFaultShadow
10: fffff8011dd6c880 nt!KiFloatingErrorFaultShadow
11: fffff8011dd6c900 nt!KiAlignmentFaultShadow

(...)
```

Обратите внимание на имена функций — в основном их смысл понятен без комментариев. Эти элементы таблицы связаны с ошибками Intel/AMD (в данном примере). Несколько распространенных примеров исключений:

- Деление на нуль (0).
- Точка прерывания (3) — обрабатывается ядром прозрачно, при этом управление передается присоединенному отладчику (если он есть).
- Недопустимый код операции (6) — ошибка выдается процессором при обнаружении неизвестной команды.
- Ошибка страницы (14) — выдается процессором, если у элемента таблицы страниц, используемой для преобразования виртуальных адресов в физические, бит `Valid` равен 0; это означает, что (по мнению процессора) страница не находится в физической памяти.

Другие исключения выдаются ядром в результате предыдущей ошибки процессора. Например, при возникновении ошибки страницы обработчик ошибок страниц диспетчера памяти попытается найти адреса, отсутствующие в физической памяти. Если окажется, что страница не существует вообще, диспетчер памяти выдаст исключение нарушения прав доступа.

После выдачи исключения ядро ищет функцию, в которой произошло исключение для обработчика (кроме определенных исключений, которые он обрабатывает прозрачно, например точек прерывания (3)). Если функция не будет найдена, поиск переходит вверх по стеку вызовов, пока обработчик не будет найден. Если стек вызовов будет пройден полностью, в системе происходит фатальный сбой.

Как драйвер может обрабатывать исключения этих типов? Компания Microsoft добавила в язык C четыре ключевых слова, упрощающих обработку таких исключений. Новые ключевые слова с краткими описаниями перечислены в табл. 6.1.

Таблица 6.1. Ключевые слова для работы с SEH

Ключевое слово	Описание
`__try`	Начинает блок кода, в котором могут происходит исключения
`__except`	Указывает, обрабатывается ли исключение, и если обрабатывается, предоставляет код обработки
`__finally`	Не связано с исключениями напрямую. Предоставляет код, который гарантированно будет выполнен в любом случае, по какой бы причине ни произошел выход из блока `__try` — нормально или из-за исключения
`__leave`	Предоставляет оптимизированный механизм для перехода к блоку __finally из любой точки блока `__try`

Допустимые комбинации ключевых слов — `__try`/`__except` и `__try`/`__finally`. Тем не менее они могут объединяться посредством вложения до любого уровня.

Эти ключевые слова также работают в пользовательском режиме практически по тем же правилам.

Использование __try/__except

В главе 4 мы реализовали драйвер, который обращается к буферу пользовательского режима для получения данных, необходимых для работы драйвера. При этом использовался прямой указатель на пользовательский буфер. Тем не менее безопасность такого решения не гарантирована. Например, код пользовательского режима (допустим, из другого потока) может освободить буфер непосредственно перед обращением к нему из драйвера. В таких случаях драйвер вызовет фатальный сбой системы, что, по сути, объясняется ошибкой пользователя (или злым умыслом). Так как пользовательским данным доверять нельзя, такие обращения должны заключаться в блок `__try`/`__except`, чтобы некорректный буфер не привел к сбою драйвера.

Важная часть переработанного обработчика `IRP_MJ_DEVICE_CONTROL`, использующая обработчик исключений:

```
case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
{
    if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(ThreadData))
{
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    auto data = (ThreadData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
    if (data == nullptr) {
        status = STATUS_INVALID_PARAMETER;
```

```
        break;
    }
    __try {
        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        PETHREAD Thread;
        status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId),
&Thread); if (!NT_SUCCESS(status))
            break;
        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);
        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // Какие-то проблемы с буфером
        status = STATUS_ACCESS_VIOLATION;
    }
    break;
}
```

Включение `EXCEPTION_EXECUTE_HANDLER` в `__except` означает, что любое исключение должно быть обработано. Можно действовать более избирательно — вызвать `GetExceptionCode` и проверить фактическое исключение. В противном случае можно приказать ядру продолжить поиск обработчика по стеку вызовов:

```
__except (GetExceptionCode() == STATUS_ACCESS_VIOLATION
    ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // Обработка исключения
}
```

Означает ли все это, что драйвер может перехватывать любые исключения? В таком случае драйвер никогда не вызовет фатального сбоя системы. К сожалению (или к счастью — зависит от точки зрения), это не так.

Например, нарушение прав доступа может быть перехвачено только в том случае, если адрес нарушения находится в пользовательском пространстве. Если он находится в пространстве режима ядра, то исключение не будет перехвачено и произойдет фатальный сбой системы. И это разумно, потому что произошло нечто очень плохое, и ядро не должно разрешить драйверу скрыть этот факт.

С другой стороны, адреса пользовательского режима не находятся под контролем драйвера, поэтому такие исключения могут быть перехвачены и обработаны.

✦ Механизм SEH также может использоваться драйверами (и кодом пользовательского режима) для выдачи нестандартных исключений.

- Ядро предоставляет обобщенную функцию `ExRaiseStatus` для выдачи любого исключения, а также ряд специализированных функций (например, `ExRaiseAccessViolation`).

Возможно, вас интересует, как работают исключения в языках высокого уровня (C++, Java, .NET и т. д.). Разумеется, это зависит от реализации, но компиляторы Microsoft используют SEH со своими кодами для конкретной платформы. Например, исключения .NET используют значение `0xe0434c52`. Это значение было выбрано из-за того, что последние 3 байта являются ASCII-кодами текста 'CLR'. 0xe0 присутствует только для того, чтобы код случайно не совпал с другим кодом исключения. Команда C# `throw` использует функцию API `RaiseException` в пользовательском режиме для того, чтобы выдать это исключение с аргументами, которые предоставляют среде .NET CLR (Common Language Runtime) необходимую информацию для распознавания выданного объекта исключения.

Использование __try/__finally

Блок `__try` и `__finally` не имеет прямого отношения к обработке исключений. Он скорее гарантирует, что некоторый блок кода будет выполнен в любом случае — как при корректном завершении кода, так и при преждевременном выходе из-за исключения. На концептуальном уровне этот блок напоминает ключевое слово `finally`, встречающееся во многих высокоуровневых языках (например, Java и C#). Простой пример для демонстрации проблемы:

```
void foo() {
    void* p = ExAllocatePool(PagedPool, 1024);
    // Выполнить действия с p
    ExFreePool(p);
}
```

Этот код выглядит достаточно безобидно. Тем не менее он скрывает ряд проблем:

- Если между выделением памяти и ее освобождением произойдет исключение, будет проведен поиска обработчика на стороне вызова, но память не освободится.
- Если команда `return` будет выполнена в некоторой условной конструкции между выделением и освобождением памяти, то буфер не будет освобожден. Разработчик должен действовать осторожно и следить за тем, чтобы все ветви выхода из функции проходили через команду освобождения буфера.

Второй пункт может быть реализован посредством особенно тщательного программирования, но это лишнее бремя, которого лучше избегать. Первый пункт

стандартными средствами программирования вообще не решается. На помощь приходит конструкция `__try/__finally`. Она гарантирует, что буфер будет освобожден при любом развитии событий:

```
void foo() {
    void* p = ExAllocatePool(PagedPool, 1024);
    __try {
        // Выполнить действия с p
    }
    __finally {
        ExFreePool(p);
    }
}
```

Даже при том, что команды `return` вроде бы находятся в теле `__try`, код `__finally` будет вызван до фактического возврата из функции. Если произойдет исключение, то блок `__finally` будет выполнен до того, как ядро начнет поиск обработчиков по стеку вызовов.

Конструкция `__try/__finally` пригодится не только для выделения памяти, но и для других ресурсов, с которыми используются операции захвата и освобождения. Распространенный пример — синхронизация потоков, обращающихся к некоторым общим данным. Пример захвата и освобождения быстрого мьютекса (быстрые мьютексы и другие примитивы синхронизации описаны позднее в этой главе):

```
FAST_MUTEX MyMutex;
void foo() {
    ExAcquireFastMutex(&MyMutex);
    __try {
        // Выполнить некоторую работу,
        // пока удерживается быстрый мьютекс
    }
    __finally {
        ExReleaseFastMutex(&MyMutex);
    }
}
```

Использование RAII-оберток C++ вместо __try/__finally

Хотя приведенные примеры с `__try/__finally` и работают, использовать их неудобно. На языке C++ можно строить RAII-обертки, которые обеспечивают нужный результат без использования `__try/__finally`. В языке C++ нет ключевого слова `finally`, в отличие от C# или Java, но оно ему и не нужно — есть деструкторы.

Рассмотрим очень простой, минимальный пример, управляющий выделением памяти при помощи RAII-обертки:

```
template<typename T = void>
struct kunique_ptr {
    kunique_ptr(T* p = nullptr) : _p(p) {}
    ~kunique_ptr() {
        if (_p)
            ExFreePool(_p);
    }

    T* operator->() const {
        return _p;
    }

    T& operator*() const {
        return *_p;
    }

private:
    T* _p;
};
```

Класс использует шаблоны, упрощающие работу с любыми типами данных. Пример использования:

```
struct MyData {
    ULONG Data1;
    HANDLE Data2;
};

void foo() {
    // Выделение памяти
    kunique_ptr<MyData> data((MyData*)ExAllocatePool(PagedPool, sizeof(MyData)));
    // Использование указателя
    data->Data1 = 10;
    // При выходе объекта из области видимости
    // дескриптор освобождает буфер.
}
```

> Если C++ не является вашим основным языком программирования, то приведенный фрагмент может показаться непонятным. В таком случае можно продолжать использовать `__try`/`__finally`, но я бы порекомендовал привыкнуть к такому коду. В любом случае, даже если приведенная реализация `kunique_ptr` кажется вам непонятной, вы можете пользоваться ею без понимания всех мелких подробностей.

Представленный тип `kunique_ptr` содержит минимум возможностей. Вам также следует удалить копирующий конструктор и присваивание с копированием,

и, возможно, разрешить присваивание с перемещением (C++ 11 и выше) и другие вспомогательные возможности. Более полная реализация:

```
template<typename T = void>
struct kunique_ptr {
    kunique_ptr(T* p = nullptr) : _p(p) {}

    // разрешить передачу владения
    kunique_ptr(const kunique_ptr&) = delete;
    kunique_ptr& operator=(const kunique_ptr&) = delete;

    // разрешить передачу владения
    kunique_ptr(kunique_ptr&& other) : _p(other._p) {
        other._p = nullptr;
    }

    kunique_ptr& operator=(kunique_ptr&& other) {
        if (&other != this) {
            Release();
            _p = other._p;
            other._p = nullptr;
        }
        return *this;
    }

    ~kunique_ptr() {
        Release();
    }

    operator bool() const {
        return _p != nullptr;
    }

    T* operator->() const {
        return _p;
    }

    T& operator*() const {
        return *_p;
    }

    void Release() {
        if (_p)
        ExFreePool(_p);
    }

private:
    T* _p;
};
```

Позднее в этой главе будут построены другие RAII-обертки для примитивов синхронизации.

Фатальный сбой

Как вы уже знаете, при возникновении необработанного исключения в режиме ядра в системе происходит фатальный сбой; чаще всего вы видите «синий экран смерти» (BSOD). В этом разделе речь пойдет о том, что происходит при сбое системы и что делать в таких случаях.

Фатальный сбой системы известен под разными именами — «синий экран смерти» (BSOD), «общий отказ», «стоп-ошибка», но все они означают одно и то же. BSOD — не наказание, как может показаться на первый взгляд, а механизм защиты. Если в режиме ядра, который должен пользоваться доверием, произошло нечто плохое, вероятно, безопаснее всего будет немедленно все остановить. Если разрешить коду и дальше творить неизвестно что, это может привести к тому, что система перестанет загружаться из-за повреждения важных файлов или разделов реестра.

> В последних версиях Windows 10 для обозначения фатальных сбоев системы используются другие цвета. Для внутренних сборок применяется зеленый цвет, а я также сталкивался с оранжевым.

Если к системе, в которой произошел сбой, подключен отладчик ядра, то управление передается отладчику. Это позволяет проанализировать состояние системы до выполнения каких-либо других действий.

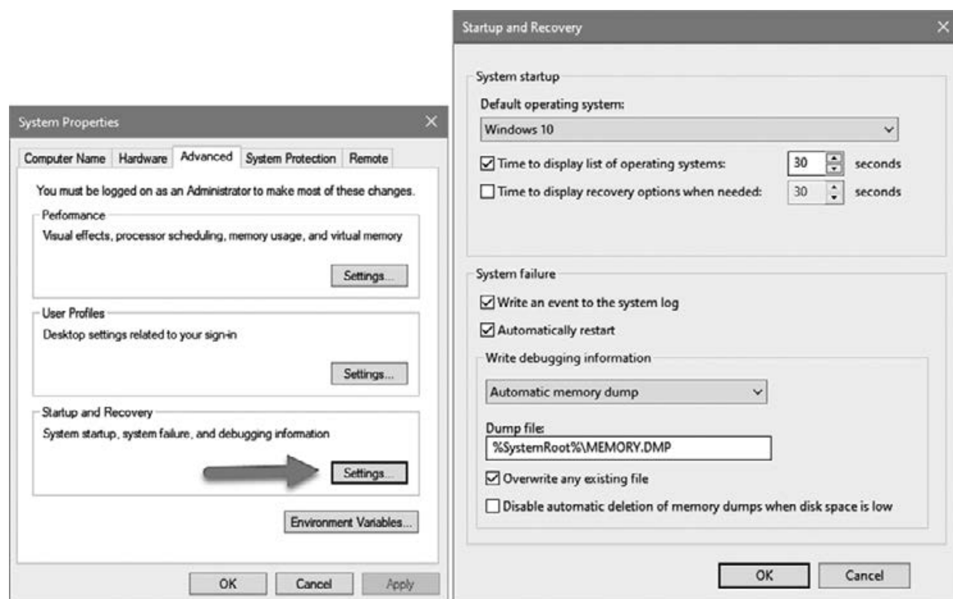


Рис. 6.7. Настройки запуска и восстановления

Систему можно настроить на выполнение определенных операций в случае сбоя системы. Это можно сделать в группе System Properties UI на вкладке Advanced. Кнопка Settings... в разделе Startup and Recovery открывает диалоговое окно Startup and Recovery; в разделе System Failure этого окна перечислены доступные варианты. На рис. 6.7 показаны оба диалоговых окна.

Если в системе происходит сбой, информация о событии может быть записана в журнал событий. По умолчанию этот режим включен (Write an event to the system log), и обычно нет веских причин для его отключения. Система настроена на автоматический перезапуск (Automatic restart); этот режим используется по умолчанию с Windows 2000.

Самая важная настройка — запись дампа (Automatic memory dump). В файле дампа сохраняется информация о состоянии на момент сбоя. Позднее эту информацию можно проанализировать, загрузив файл дампа в отладчике. Тип файла дампа также важен, так как он определяет состав информации, хранящейся в дампе. Важно подчеркнуть, что дамп не записывается в указанный файл в момент сбоя. Вместо этого он записывается в первый страничный файл. Только при перезапуске системы ядро замечает, что в страничном файле хранится информация дампа, и копирует данные в целевой файл. Такое решение было выбрано из-за того, что в момент фатального сбоя системы может быть слишком опасно записывать что-либо в новый файл; система может быть недостаточно стабильной. Лучше записать данные в страничный файл, который все равно уже открыт. С другой стороны, страничный файл должен быть достаточно большим для хранения дампа, иначе файл дампа записан не будет.

Тип дампа определяет, какие данные будут сохранены, и дает некоторое представление о размере файла. Возможные варианты:

- Small memory dump — минимальный дамп с базовой информацией о системе и потоке, вызвавшем сбой. Обычно этого недостаточно, для того чтобы определить причину происходящего (кроме самых тривиальных случаев). С другой стороны, файл имеет минимальный размер.
- Kernel memory dump — используется по умолчанию в Windows 7 и более ранних версиях. В этом режиме сохраняется вся память ядра, но не память пользовательского режима. Обычно этого оказывается достаточно, так как фатальный сбой системы может быть вызван только некорректным поведением кода режима ядра. Крайне маловероятно, что код пользовательского режима имеет к этому хоть какое-то отношение.
- Complete memory dump — сохраняется полный дамп всей памяти пользовательского режима и режима ядра, то есть самая полная доступная информация. Недостатком является размер дампа, который может быть гигантским в зависимости от объема ОЗУ системы и текущего использования памяти.

- Automatic memory dump (Windows 8+) — используется по умолчанию в Windows 8 и более поздних версиях. То же, что дамп памяти ядра, но при загрузке ядро изменяет размер страничного файла до значения, которое с высокой вероятностью гарантирует, что размер страничного файла будет достаточным для хранения дампа ядра. Это происходит только в том случае, если размер страничного файла задается значением System managed.
- Active memory dump (Windows 10+) — аналог полного дампа памяти, кроме того что если в засбоившей системе были размещены гостевые виртуальные машины, используемая ими память на момент сбоя не сохраняется. Это способствует сокращению размера файла дампа в серверных системах с большим количеством VM.

Информация дампа

Если после фатального сбоя в системе был сохранен дамп, откройте его в WinDbg командой File ▸ Open Dump File и найдите нужный файл. В отладчике выводится базовая информация следующего вида:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [C:\Temp\MEMORY.DMP]
Kernel Bitmap Dump File: Kernel address space is available, User address space\
may not be available.

************* Path validation summary **************
Response                         Time (ms)     Location
                                               SRV*c:\Symbols*http://msdl.
microsoft.\
com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff803`70abc000 PsLoadedModuleList = 0xfffff803`70eff2d0
Debug session time: Wed Apr 24 15:36:55.613 2019 (UTC + 3:00)
System Uptime: 0 days 0:05:38.923
Loading Kernel Symbols
....................................Page 2001b5efc too large to be in the dump\
file.
Page 20001ebfb too large to be in the dump file.
...............................
Loading User Symbols
PEB is paged out (Peb.Ldr = 00000054`34256018). Type ".hh dbgerr001" for details
Loading unloaded module list
```

```
.............
For analysis of this file, run !analyze -v
nt!KeBugCheckEx:
fffff803`70c78810 48894c2408      mov      qword ptr [rsp+8],rcx\
ss:fffff988`53b0f6b0\
=000000000000000a
```

Отладчик рекомендует выполнить команду `!analyze -v` — пожалуй, это самое частое, что делается в начале анализа дампа. Обратите внимание: стек вызовов находится на `KeBugCheckEx` — функции, генерирующей фатальный сбой и полностью документированной в WDK, так как драйверы также могут использовать ее при необходимости.

Стандартная логика команды `!analyze -v` выполняет базовый анализ потока, вызвавшего сбой, и выводит информацию, относящуюся к коду дампа:

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high. This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: ffffd907b0dc7660, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff80375261530, address which referenced memory

Debugging Details:
------------------

(...)

DUMP_TYPE: 1

BUGCHECK_P1: ffffd907b0dc7660

BUGCHECK_P2: 2

BUGCHECK_P3: 0

BUGCHECK_P4: fffff80375261530

READ_ADDRESS: Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool
 ffffd907b0dc7660 Paged pool
```

```
CURRENT_IRQL: 2

FAULTING_IP:
myfault+1530
fffff803`75261530 8b03 mov eax,dword ptr [rbx]

(...)

ANALYSIS_VERSION: 10.0.18317.1001 amd64fre

TRAP_FRAME: fffff98853b0f7f0 -- (.trap 0xfffff98853b0f7f0)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=ffffd90797400340
rdx=0000000000000880 rsi=0000000000000000 rdi=0000000000000000
rip=fffff80375261530 rsp=fffff98853b0f980 rbp=0000000000000002
 r8=ffffd9079c5cec10 r9=0000000000000000 r10=ffffd907974002c0
r11=ffffd907b0dc1650 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei ng nz na po nc
myfault+0x1530:
fffff803`75261530 8b03            mov eax,dword ptr [rbx] ds:00000000`00000000=?\
???????
Resetting default scope

LAST_CONTROL_TRANSFER: from fffff80370c8a469 to fffff80370c78810

STACK_TEXT:
fffff988`53b0f6a8 fffff803`70c8a469 : 00000000`0000000a
                  ffffd907`b0dc7660 00000000`0\
0000002 00000000`00000000 : nt!KeBugCheckEx
fffff988`53b0f6b0 fffff803`70c867a5 : ffff8788`e4604080
                  ffffff4c`c66c7010 00000000`0\
0000003 00000000`00000880 : nt!KiBugCheckDispatch+0x69
fffff988`53b0f7f0 fffff803`75261530 : ffffff4c`c66c7000 00000000`00000000
                  fffff988`5\
3b0f9e0 00000000`00000000 : nt!KiPageFault+0x465
fffff988`53b0f980 fffff803`75261e2d : fffff988`00000000 00000000`00000000
                  ffff8788`e\
c7cf520 00000000`00000000 : myfault+0x1530
fffff988`53b0f9b0 fffff803`75261f88 :
                  ffffff4c`c66c7010 00000000`000000f0 00000000`0\
0000001 ffffff30`21ea80aa : myfault+0x1e2d
fffff988`53b0fb00 fffff803`70ae3da9 :
                  ffff8788`e6d8e400 00000000`00000001 00000000`8\
3360018 00000000`00000001 : myfault+0x1f88
fffff988`53b0fb40 fffff803`710d1dd5 : fffff988`53b0fec0
                  ffff8788`e6d8e400 00000000`0\
0000001 ffff8788`ecdb6690 : nt!IofCallDriver+0x59
fffff988`53b0fb80 fffff803`710d172a :
                  ffff8788`00000000 00000000`83360018 00000000`0\
0000000 fffff988`53b0fec0 : nt!IopSynchronousServiceTail+0x1a5
fffff988`53b0fc20 fffff803`710d1146 : 00000054`344feb28
                                      00000000`00000000 00000000`0\
```

```
0000000 00000000`00000000 : nt!IopXxxControlFile+0x5ca
fffff988`53b0fd60 fffff803`70c89e95 : ffff8788`e4604080 fffff988`53b0fec0\
00000054`344feb28 fffff988`569fd630 : nt!NtDeviceIoControlFile+0x56
fffff988`53b0fdd0 00007ff8`ba39c147 : 00000000`00000000 00000000`00000000\
00000000`00000000 00000000`00000000 : nt!KiSystemServiceCopyEnd+0x25
00000054`344feb48 00000000`00000000 : 00000000`00000000 00000000`00000000\
00000000`00000000 00000000`00000000 : 0x00007ff8`ba39c147

(...)

FOLLOWUP_IP:
myfault+1530
fffff803`75261530 8b03            mov eax,dword ptr [rbx]

FAULT_INSTR_CODE: 8d48038b

SYMBOL_STACK_INDEX: 3

SYMBOL_NAME: myfault+1530

FOLLOWUP_NAME: MachineOwner

MODULE_NAME: myfault

IMAGE_NAME: myfault.sys

(...)
```

Каждый код дампа может содержать до четырех чисел, предоставляющих дополнительную информацию о сбое. В данном случае мы видим код `DRIVER_IRQL_NOT_LESS_OR_EQUAL` (`0xd1`), а следующие четыре числа с именами от `Arg1` до `Arg4` означают (в этой последовательности): память, уровень IRQL на момент вызова, операция чтения или записи и адрес, с которого поступило обращение.

Команда четко распознает myfault.sys как сбойный модуль (драйвер). В данном случае мы имеем дело с простым сбоем — виновник находится на вершине стека вызовов, как видно из раздела `STACK TEXT` (также можно воспользоваться командой k, чтобы снова просмотреть информацию).

Команда `!analyze -v` может расширяться; для добавления дополнительных аналитических возможностей используются DLL-библиотеки расширения. Некоторые расширения можно найти в интернете.

За дополнительной информацией о добавлении собственных возможностей анализа к команде обращайтесь к документации API отладчика.

В более сложных дампах могут отображаться вызовы режима ядра только из стека вызовов потока-нарушителя. Прежде чем делать вывод, что вы об-

наружили ошибку в ядре Windows, учтите следующее: скорее всего, драйвер сделал что-то такое, что само по себе не фатально (например, произошло переполнение буфера — данные были записаны за границей выделенного буфера), но, к сожалению, память за буфером была выделена другим драйвером или ядром, и ничего плохого в тот момент не произошло. Через какое-то время ядро обратилось к этой памяти, получило некорректные данные и вызвало фатальный сбой системы. Однако драйвер, который стал причиной нарушения, уже не находится в стеке вызовов; диагностировать такие ошибки намного сложнее.

Для диагностики таких проблем может использоваться Driver Verifier. Основы работы с Driver Verifier будут рассмотрены в главе 11.

После получения кода дампа будет полезно заглянуть в документацию отладчика. В разделе «Bugcheck Code Reference» наиболее распространенные коды ошибок рассмотрены более подробно, с указанием типичных причин и советами относительно того, где продолжить поиски.

Анализ файла дампа

Файл дампа содержит «мгновенный снимок» системы. В остальном файлы дампа остаются неизменными для всех сеансов отладки ядра. Вы не можете устанавливать точки прерывания и, конечно, не можете использовать команды `go`. Все остальные команды доступны в обычном виде. Такие команды, как `!process`, `!thread`, `lm` и `k`, используются как обычно. Ниже перечислены другие команды и рекомендации:

- В приглашении указан текущий процессор. Переключение процессоров может осуществляться командой `~`*n*`s`, где *n* — индекс процессора (по аналогии с переключением потоков в пользовательском режиме).
- Команда `!running` выводит список потоков, выполнявшихся на всех процессорах на момент сбоя. С добавлением параметра `-t` выводится стек вызовов для каждого потока. Пример для приведенного выше дампа:

```
2: kd> !running -t

System Processors: (000000000000000f)
    Idle Processors: (0000000000000002)

         Prcbs             Current         (pri) Next        (pri)    Idle
  0     fffff8036ef3f180  ffff8788e91cf080 ( 8) fffff8037104840\
```

```
0 ................

 # Child-SP          RetAddr           Call Site
00 00000094`ed6ee8a0 00000000`00000000 0x00007ff8`b74c4b57

  2     ffffb000c1944180   ffff8788e4604080 (12)
ffffb000c195514\
0 ................

 # Child-SP          RetAddr           Call Site
00 fffff988`53b0f6a8 fffff803`70c8a469 nt!KeBugCheckEx
01 fffff988`53b0f6b0 fffff803`70c867a5 nt!KiBugCheckDispatch+0x69
02 fffff988`53b0f7f0 fffff803`75261530 nt!KiPageFault+0x465
03 fffff988`53b0f980 fffff803`75261e2d myfault+0x1530
04 fffff988`53b0f9b0 fffff803`75261f88 myfault+0x1e2d
05 fffff988`53b0fb00 fffff803`70ae3da9 myfault+0x1f88
06 fffff988`53b0fb40 fffff803`710d1dd5 nt!IofCallDriver+0x59
07 fffff988`53b0fb80 fffff803`710d172a nt!IopSynchronousServiceTail+0x1a5
08 fffff988`53b0fc20 fffff803`710d1146 nt!IopXxxControlFile+0x5ca
09 fffff988`53b0fd60 fffff803`70c89e95 nt!NtDeviceIoControlFile+0x56
0a fffff988`53b0fdd0 00007ff8`ba39c147 nt!KiSystemServiceCopyEnd+0x25
0b 00000054`344feb48 00000000`00000000 0x00007ff8`ba39c147

  3 ffffb000c1c80180 ffff8788e917e0c0 ( 5) ffffb000c1c9114\
0 ................

 # Child-SP          RetAddr           Call Site
00 fffff988`5683ec38 fffff803`70ae3da9 Ntfs!NtfsFsdClose
01 fffff988`5683ec40 fffff803`702bb5de nt!IofCallDriver+0x59
02 fffff988`5683ec80 fffff803`702b9f16 FLTMGR!FltpLegacyProcessingAfterPre\
CallbacksCompleted+0x15e
03 fffff988`5683ed00 fffff803`70ae3da9 FLTMGR!FltpDispatch+0xb6
04 fffff988`5683ed60 fffff803`710cfe4d nt!IofCallDriver+0x59
05 fffff988`5683eda0 fffff803`710de470 nt!IopDeleteFile+0x12d
06 fffff988`5683ee20 fffff803`70aea9d4 nt!ObpRemoveObjectRoutine+0x80
07 fffff988`5683ee80 fffff803`723391f5 nt!ObfDereferenceObject+0xa4
08 fffff988`5683eec0 fffff803`72218ca7 Ntfs!NtfsDeleteInternalAttributeStream+\
0x111
09 fffff988`5683ef00 fffff803`722ff7cf Ntfs!NtfsDecrementCleanupCounts+0x147
0a fffff988`5683ef40 fffff803`722fe87d Ntfs!NtfsCommonCleanup+0xadf
0b fffff988`5683f390 fffff803`70ae3da9 Ntfs!NtfsFsdCleanup+0x1ad
0c fffff988`5683f6e0 fffff803`702bb5de nt!IofCallDriver+0x59
0d fffff988`5683f720 fffff803`702b9f16 FLTMGR!FltpLegacyProcessingAfter\
PreCallbac\ksCompleted+0x15e
0e fffff988`5683f7a0 fffff803`70ae3da9 FLTMGR!FltpDispatch+0xb6
0f fffff988`5683f800 fffff803`710ccc38 nt!IofCallDriver+0x59
10 fffff988`5683f840 fffff803`710d4bf8 nt!IopCloseFile+0x188
11 fffff988`5683f8d0 fffff803`710d9f3e nt!ObCloseHandleTableEntry+0x278
12 fffff988`5683fa10 fffff803`70c89e95 nt!NtClose+0xde
13 fffff988`5683fa80 00007ff8`ba39c247 nt!KiSystemServiceCopyEnd+0x25
14 000000b5`aacf9df8 00000000`00000000 0x00007ff8`ba39c247
```

Результат неплохо показывает, что происходило в момент сбоя.

- По умолчанию команда `!stacks` выводит стеки всех потоков. Другая, более полезная разновидность — строка поиска, с которой выводятся только те потоки, в которых присутствует модуль или функция, содержащие эту строку. Это позволяет найти код драйвера в системе (потому что он мог не выполняться в момент сбоя, но присутствовать в стеке вызова некоторого потока). Пример для приведенного выше дампа:

```
2: kd> !stacks
Proc.Thread  .Thread  Ticks   ThreadState Blocker
                            [fffff803710459c0 Idle]
   0.000000 fffff80371048400 0000003 RUNNING nt!KiIdleLoop+0x15e
   0.000000 ffffb000c17b1140 0000ed9 RUNNING hal!HalProcessorIdle+0xf
   0.000000 ffffb000c1955140 0000b6e RUNNING nt!KiIdleLoop+0x15e
   0.000000 ffffb000c1c91140 000012b RUNNING nt!KiIdleLoop+0x15e
                           [ffff8788d6a81300 System]
   4.000018 ffff8788d6b8a080 0005483 Blocked nt!PopFxEmergencyWorker+0x3e
   4.00001c ffff8788d6bc5140 0000982 Blocked nt!ExpWorkQueueManagerThread+0x127
   4.000020 ffff8788d6bc9140 000085a Blocked nt!KeRemovePriQueue+0x25c

(...)

2: kd> !stacks 0 myfault
Proc.Thread  .Thread  Ticks   ThreadState Blocker
                            [fffff803710459c0 Idle]
                            [ffff8788d6a81300 System]

(...)

                            [ffff8788e99070c0 notmyfault64.e]
af4.00160c  ffff8788e4604080 0000006 RUNNING     nt!KeBugCheckEx

(...)
```

Адрес рядом с каждой строкой — адрес структуры `ETHREAD` потока, которая может передаваться команде `!thread`.

Зависание системы

Системный сбой — самый распространенный тип дампа, который обычно анализируется разработчиком. Тем не менее существует еще одна разновидность дампов, с которой вам, возможно, придется работать: зависание системы. Зависшая система ни на что не реагирует (или почти не реагирует). Выполнение почему-то остановилось или попало в состояние взаимной блокировки. Никакого фатального сбоя в системе не происходит, поэтому прежде всего необходимо понять — как получить дамп системы?

Файл дампа содержит некоторое состояние системы, он не обязан быть связан с фатальным сбоем или другим аномальным состоянием. Существуют специальные инструменты (включая отладчик режима ядра), которые позволяют сгенерировать файл дампа в любой момент времени.

Если система еще продолжает проявлять признаки жизни, программа NotMyFault из пакета Sysinternals может форсировать фатальный сбой системы и обеспечить генерирование файла дампа (собственно, дамп из предыдущего раздела был сгенерирован именно таким способом). На рис. 6.8 изображен скриншот NotMyFault. Если выбрать первый вариант (по умолчанию) и щелкнуть на кнопке Crash, в системе немедленно происходит фатальный сбой, для которого генерируется файл дампа (если эта возможность была включена в настройках).

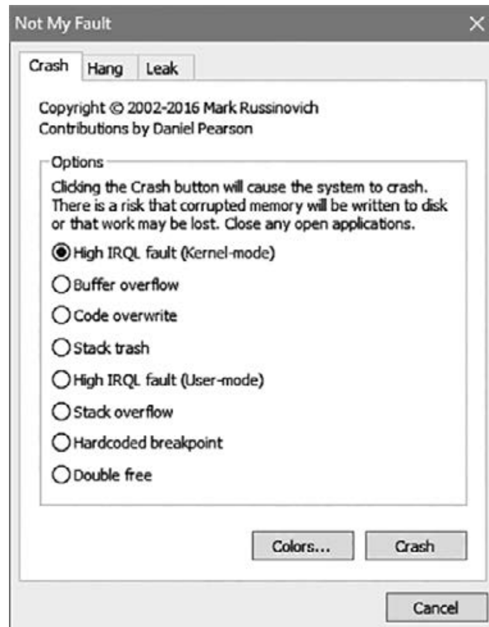


Рис. 6.8. NotMyFault

NotMyFault использует драйвер myfault.sys, который несет непосредственную ответственность за сбой.

NotMyFault существует в 32- и 64-разрядных версиях (имя файла завершается суффиксом «64»). Не забудьте выбрать версию, соответствующую вашей системе, в противном случае драйвер не загрузится.

Если система полностью перестала реагировать и вы можете присоединить отладчик ядра (управляемая машина была настроена для отладки), проводите отладку обычным образом или сгенерируйте файл дампа командой `.dump`.

Если система полностью перестала реагировать, но отладчик ядра присоединить не удается, фатальный сбой можно сгенерировать вручную, если соответствующее изменение было внесено в реестр заранее (то есть вы заранее предполагали, что система зависнет). При обнаружении определенной комбинации клавиш драйвер клавиатуры генерирует фатальный сбой. За полной информацией обращайтесь по ссылке[1]. В этом случае используется код сбоя 0xe2 (`MANUALLY_INITIATED_CRASH`).

Синхронизация потоков

Потокам иногда приходится координировать свою работу. Классический пример — драйвер, использующий связный список для сбора элементов данных. Драйвер может вызываться несколькими клиентами из разных потоков одного или нескольких процессов. Это означает, что операции со связным списком должны выполняться атомарно, чтобы список не был случайно поврежден. Если несколько потоков обратятся к одному блоку памяти и по крайней мере один из них вносит изменения, возникает ситуация *гонки по данным*. При возникновении гонки по данным ничего предсказать нельзя — может произойти все что угодно. Как правило, в драйвере рано или поздно произойдет фатальный сбой системы; повреждение данных практически гарантировано.

В таких ситуациях очень важно, чтобы пока один поток работает со связным списком, все остальные потоки отступили и каким-то образом дождались, пока первый поток завершит свою работу. Только тогда другой поток (всего один) сможет изменять список. Это пример так называемой *синхронизации потоков*.

Ядро предоставляет ряд примитивов, упрощающих синхронизацию для защиты данных при параллельном доступе. В этом разделе рассматриваются различные примитивы и приемы, используемые для синхронизации потоков.

Операции со взаимоблокировкой

Функции со взаимоблокировкой (такие функции имеют префикс `Interlocked`) предоставляют удобные операции, которые выполняются атомарно на аппаратном уровне (то есть программные объекты при этом не используются). Если

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/forcing-a-system-crash-from-the-keyboard.

ваша задача решается с этими функциями, используйте их, потому что это самое эффективное решение.

Простой пример — инкремент (увеличение целого числа на 1). В общем случае эти операции не являются атомарными. Если два (и более) потока попытаются выполнить операцию одновременно с одним адресом памяти, возможно (и даже вероятно), некоторые увеличения будут потеряны. На рис. 6.9 изображен простой сценарий, в котором при увеличении значения из двух потоков будет получен результат 1 вместо 2.

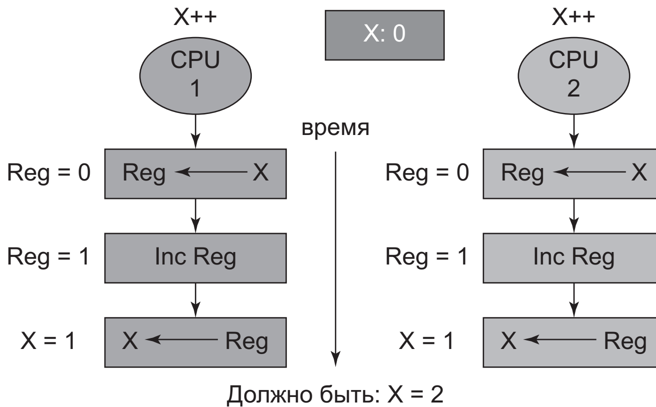


Рис. 6.9. Конкурентное выполнение инкремента

Пример на рис. 6.9 сильно упрощен. На настоящих процессорах приходится учитывать другие эффекты, особенно кэширование, с которым этот сценарий становится еще более вероятным. Кэширование процессоров, буферизация и другие аспекты современных процессоров — нетривиальная тема, выходящая далеко за рамки книги.

В табл. 6.2 перечислены некоторые функции `Interlocked`, доступные для использования в драйверах.

Таблица 6.2. Некоторые функции Interlocked

Функция	**Описание**
`InterlockedIncrement` / `InterlockedIncrement16` / `InterlockedIncrement64`	Атомарное увеличение 32/16/64-разрядного числа на 1
`InterlockedDecrement` / `16` / `64`	Атомарное уменьшение 32/16/64-разрядного числа на 1

Функция	Описание
InterlockedAdd / InterlockedAdd64	Атомарное прибавление 32/64-разрядного числа к переменной
InterlockedExchange / 8 / 16 / 64	Атомарная перестановка двух 32/8/16/64-разрядных значений
InterlockedCompareExchange / 64 / 128	Атомарное сравнение переменной со значением. В случае равенства функция заменяет значение предоставленным и возвращает TRUE; в остальных случаях функция помещает текущее значение в переменную и возвращает FALSE

Семейство функций `InterlockedCompareExchange` используется в программировании без блокировок — методологии программирования, предназначенной для выполнения сложных атомарных операций без использования программных объектов. Эта тема выходит за рамки книги.

Функции из табл. 6.2 также доступны в пользовательском режиме, так как на самом деле это не функции, а специальные команды процессора.

Объекты диспетчеризации

Ядро предоставляет набор примитивов, называемых *объектами диспетчеризации*, или *ожидаемыми* (waitable) объектами. Эти объекты могут находиться в установленном (signaled) и сброшенном (non-signaled) состоянии, причем сам смысл этих терминов зависит от типа объекта. Объекты называются «ожидаемыми», потому что поток может перейти в ожидание до того момента, когда объект перейдет в установленное состояние. Во время ожидания поток не потребляет процессорное время, так как он находится в состоянии *ожидания*.

Основные функции, используемые при ожидании, — `KeWaitForSingleObject` и `KeWaitForMultipleObjects`. Ниже приведены их прототипы (с упрощенными аннотациями SAL для ясности):

```
NTSTATUS KeWaitForSingleObject (
    _In_ PVOID Object,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout);
NTSTATUS KeWaitForMultipleObjects (
    _In_ ULONG Count,
    _In_reads_(Count) PVOID Object[],
```

```
    _In_ WAIT_TYPE WaitType,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout,
    _Out_opt_ PKWAIT_BLOCK WaitBlockArray);
```

Краткая сводка аргументов функций:

- `Object` — объект, используемый для ожидания. Обратите внимание: эти функции работают с объектами, а не с дескрипторами. Если у вас имеется дескриптор (возможно, переданный из пользовательского режима), вызовите `ObReferenceObjectByHandle` для получения указателя на объект.
- `WaitReason` — причина ожидания. Список возможных причин достаточно велик, но драйверы должны обычно выбирать значение `Executive`, кроме случаев ожидания по запросу пользователя, в которых выбирается значение `UserRequest`.
- `WaitMode` — режим ожидания; допустимые значения — `UserMode` или `KernelMode`. В драйверах обычно должно задаваться значение `KernelMode`.
- `Alertable` — признак того, должен ли поток находиться в тревожном состоянии во время ожидания. В тревожном состоянии возможна доставка асинхронных вызовов процедур, то есть APC-вызовов. APC-вызовы пользовательского режима могут доставляться в режиме ожидания `UserMode`. Для большинства драйверов этот аргумент должен содержать `FALSE`.
- `Timeout` — время ожидания. Если задано значение `NULL`, то ожидание не ограничивается по времени — оно продолжается до тех пор, пока объект не перейдет в установленное состояние. Значение аргумента задается в 100-наносекундных единицах, причем отрицательные числа задают относительный интервал ожидания, а положительные — абсолютное время от полуночи 1 января 1601 года.
- `Count` — количество объектов, по которым выполняется ожидание.
- `Object[]` — массив указателей на объекты, по которым выполняется ожидание.
- `WaitType` — ожидание перехода в установленное состояние всех объектов (`WaitAll`) или только одного объекта (`WaitAny`).
- `WaitBlockArray` — массив структур, используемых во внутренней реализации для управления ожиданием. Необязателен, если количество объектов `<=THREAD_WAIT_OBJECTS` (в настоящее время 3), — ядро будет использовать встроенный массив, присутствующий в каждом потоке. При большем количестве объектов драйвер должен выделить правильное количество структур из невыгружаемого пула и освободить их после завершения ожидания.

Основные возвращаемые значения `KeWaitForSingleObject`:

- `STATUS_SUCCESS` — ожидание завершено из-за перехода объекта в установленное состояние.
- `STATUS_TIMEOUT` — ожидание завершено из-за истечения тайм-аута.

Обратите внимание: все возвращаемые значения функций ожидания проходят макрос NT_SUCCESS со значением `true`.

Возвращаемые значения `KeWaitForMultipleObjects` поддерживают `STATUS_TIMEOUT`, как и возвращаемые значения `KeWaitForSingleObject`. `STATUS_SUCCESS` возвращается в том случае, если задан тип ожидания `WaitAll` и все объекты перешли в установленное состояние. Для ожидания `WaitAny` в случае перехода одного из объектов в установленное состояние возвращаемое значение представляет собой индекс в массиве объектов.

С функциями ожидания связан ряд нюансов, особенно если выбран режим ожидания UserMode, а ожидание является тревожным. За подробностями обращайтесь к документации WDK.

В табл. 6.3 перечислены основные объекты диспетчеризации и смысл установленного/сброшенного состояния для этих объектов.

Таблица 6.3. Типы объектов и смысл установленного состояния

Тип объекта	Смысл установленного состояния	Смысл сброшенного состояния
Процесс	Процесс завершился (независимо от причины)	Процесс не завершился
Поток	Поток завершился (независимо от причины)	Поток не завершился
Мьютекс	Мьютекс свободен (никем не захвачен)	Мьютекс захвачен
Событие	Событие установлено	Событие сброшено
Семафор	Счетчик семафора больше 0	Счетчик семафора равен 0
Таймер	Интервал таймера истек	Интервал таймера еще не истек
Файл	Асинхронная операция ввода/вывода завершена	Асинхронная операция ввода/вывода в процессе выполнения

Все типы объектов из табл. 6.3 также экспортируются в пользовательский режим. Основные функции ожидания в пользовательском режиме — `WaitForSingleObject` и `WaitForMultipleObjects`.

Ниже рассматриваются некоторые популярные типы объектов, используемые при синхронизации в драйверах. Также будут рассмотрены другие объекты, которые не являются объектами диспетчеризации, но поддерживают ожидание для синхронизации потоков.

Мьютекс

Мьютекс (mutex) — классический объект для решения канонической задачи: имеется набор потоков, в любой момент времени с общим ресурсом может работать только один поток из набора.

Мьютекс также иногда называется «мутантом» (mutant). Это одно и то же.

Мьютекс находится в установленном состоянии, когда он свободен. Когда поток вызывает функцию ожидания, мьютекс переходит в сброшенное состояние, а поток захватывает мьютекс (становится его владельцем). Концепция захвата очень важна для мьютекса. Она означает следующее:

- Если поток является владельцем мьютекса, только он сможет освободить мьютекс.
- Мьютекс может быть захвачен многократно одним потоком. Вторая попытка автоматически завершается успехом, так как поток является текущим владельцем мьютекса. Это также означает, что поток должен освободить мьютекс столько раз, сколько раз он был захвачен, только тогда мьютекс снова становится свободным (установленным).

Чтобы использовать мьютекс, необходимо создать в невыгружаемом пуле структуру `KMUTEX`. API мьютексов содержит следующие функции для работы с `KMUTEX`:

- Функция `KeInitializeMutex` вызывается однократно для инициализации мьютекса.
- Одна из функций ожидания, которой передается адрес созданной структуры `KMUTEX`.
- Функция `KeReleaseMutex` вызывается в тот момент, когда поток, захвативший мьютекс, хочет освободить его.

В следующем примере мьютекс используется для обращения к общим данным, чтобы они гарантированно использовались только одним потоком в любой момент времени:

```
KMUTEX MyMutex;
```

```
LIST_ENTRY DataHead;

void Init() {
    KeInitializeMutex(&MyMutex, 0);
}

void DoWork() {
    // Ожидание доступности мьютекса

    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);

    // Свободное обращение к DataHead

    // Освобождение мьютекса после завершения
    KeReleaseMutex(&MyMutex, FALSE);
}
```

Очень важно, чтобы мьютекс был освобожден независимо от того, что произойдет в программе. Рекомендуется использовать конструкцию `__try/__finally`, чтобы код освобождения был гарантированно выполнен в любых обстоятельствах:

```
void DoWork() {
    // Ожидание доступности мьютекса

    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);
    __try {
        // Свободное обращение к DataHead

    }
    __finally {
        // Освобождение мьютекса после завершения

        KeReleaseMutex(&MyMutex, FALSE);
    }
}
```

Работать с `__try/__finally` неудобно, поэтому вы можете воспользоваться C++ для создания RAII-обертки для операций ожидания. Она также может использоваться с другими примитивами синхронизации.

Сначала создается обертка для мьютекса с функциями `Lock` и `Unlock`:

```
struct Mutex {
    void Init() {
        KeInitializeMutex(&_mutex, 0);
    }

    void Lock() {
        KeWaitForSingleObject(&_mutex, Executive, KernelMode, FALSE, nullptr);
    }

    void Unlock() {
```

```
        KeReleaseMutex(&_mutex, FALSE);
    }

private:
    KMUTEX _mutex;
};
```

Теперь можно создать обобщенную RAII-обертку для ожидания по любому типу, содержащему функции `Lock` и `Unlock`:

```
template<typename TLock>
struct AutoLock {
    AutoLock(TLock& lock) : _lock(lock) {
        lock.Lock();
    }

    ~AutoLock() {
        _lock.Unlock();
    }
private:
    TLock& _lock;
};
```

С этими определениями можно заменить код использования мьютекса следующим:

```
Mutex MyMutex;

void Init() {
    MyMutex.Init();
}

void DoWork() {
    AutoLock<Mutex> locker(MyMutex);

    // Свободное обращение к DataHead
}
```

Так как блокировка должна удерживаться в течение минимально возможного времени, мы используем искусственную область видимости AutoLock, для того чтобы мьютекс захватывался как можно позднее и освобождался как можно ранее.

Начиная с C++ 17, AutoLock может использоваться без указания типа: `AutoLock locker(MyMutex);`. Так как Visual Studio в настоящее время по умолчанию использует стандарт языка C++ 14, вам придется внести соответствующие изменения в свойствах проекта в разделе C++ Language / C++ Language Standard.

Тот же тип `AutoLock` может использоваться с другими примитивами синхронизации.

Что произойдет, если мьютекс не будет освобожден потоком, а сам поток будет уничтожен? В этом случае ядро явно освободит мьютекс (так как никакой другой поток этого сделать не сможет), а следующий поток, который захватит этот мьютекс, получит в своей функции ожидания STATUS_ABANDONED (потерянный мьютекс). С точки зрения другого потока мьютекс захватывается как обычно, но код указывает на некоторую ошибку или аномальное состояние, из-за которого первый поток был завершен до освобождения мьютекса.

Быстрый мьютекс

Быстрый мьютекс — более эффективная альтернатива для классического мьютекса. Быстрый мьютекс не является объектом диспетчеризации и имеет собственный API захвата и освобождения. По сравнению с обычным мьютексом обладает следующими характеристиками:

- Быстрый мьютекс не может захватываться рекурсивно. Попытки такого рода приводят к взаимной блокировке.
- При захвате быстрого мьютекса уровень IRQL процессора повышается до APC_LEVEL (1). В результате доставка APC-вызовов в этом потоке становится невозможной.
- Ожидание по быстрому мьютексу может быть только неограниченным — задать интервал тайм-аута невозможно.

Из-за первых двух пунктов быстрые мьютексы работают немного быстрее обычных. Собственно, большинство драйверов, которым необходимы мьютексы, используют быстрые мьютексы, если только не найдется веских причин для обратного.

Быстрый мьютекс инициализируется созданием структуры FAST_MUTEX в невыгружаемом пуле и вызовом ExInitializeFastMutex. Захват быстрого мьютекса выполняется вызовом ExAcquireFastMutex или ExAcquireFastMutexUnsafe (если текущий уровень IRQL уже находится на уровне APC_LEVEL). Для освобождения быстрого мьютекса используется функция ExReleaseFastMutex или ExReleaseFastMutexUnsafe.

Быстрые мьютексы недоступны в пользовательском режиме. Код пользовательского режима может использовать только обычные мьютексы.

С точки зрения общей функциональности быстрые мьютексы эквивалентны обычным — просто они работают немного быстрее.

Мы можем создать обертку C++ для быстрого мьютекса, чтобы его захват и освобождение выполнялись автоматически при помощи класса RAII `AutoLock`, определенного в предыдущем разделе:

```
// fastmutex.h

class FastMutex {
public:
    void Init();

    void Lock();
    void Unlock();

private:
    FAST_MUTEX _mutex;
};

// fastmutex.cpp

#include "FastMutex.h"

void FastMutex::Init() {
    ExInitializeFastMutex(&_mutex);
}

void FastMutex::Lock() {
    ExAcquireFastMutex(&_mutex);
}

void FastMutex::Unlock() {
    ExReleaseFastMutex(&_mutex);
}
```

Семафор

Основное предназначение семафора — ограничение чего-либо (например, длины очереди). Семафор инициализируется максимальным и исходным значением счетчика (которому обычно присваивается максимальное значение) вызовом `KeInitializeSemaphore`. Пока значение внутреннего счетчика остается положительным, семафор установлен. Когда поток успешно вызывает `KeWaitForSingleObject`, счетчик семафора уменьшается на 1. Это продолжается до тех пор, пока счетчик не уменьшится до 0; в этот момент семафор сбрасывается.

Для примера возьмем очередь рабочих элементов, находящихся под управлением драйвера. Некоторые потоки должны добавлять элементы в очередь. Каждый такой поток вызывает `KeWaitForSingleObject` для получения одного «заряда» семафора. Пока счетчик остается положительным, поток продолжает работать и добавляет элемент в очередь, увеличивая ее длину, а поток теря-

ет один «заряд». Другие потоки занимаются обработкой рабочих элементов из очереди. Когда поток удаляет элемент из очереди, он вызывает функцию `KeReleaseSemaphore`, которая увеличивает счетчик семафора. Семафор снова переводит его в установленное состояние, позволяя другому потоку продвинуться дальше и добавить новый элемент в очередь.

При необходимости функция `KeReleaseSemaphore` может увеличить счетчик семафора более чем на 1.

Можно ли сказать, что семафор с максимальным значением 1 эквивалентен мьютексу? На первый взгляд это вроде бы так, но впечатление обманчиво. У семафора отсутствует концепция захвата; это означает, что один поток может захватить семафор, а другой — освободить его. Это достоинство, а не недостаток, как показывает приведенный пример. Семафор предназначен совершенно для иных целей, нежели мьютекс.

Событие

Событие (event) инкапсулирует флаг — переменную, которая может принимать значения `true` (установленное состояние) или `false` (сброшенное состояние). Главная цель события — сигнализировать о чем-то, произошедшем для обеспечения синхронизации потока событий. Например, если некоторое условие становится истинным, событие устанавливается, группа потоков выходит из ожидания и продолжает работать над данными, которые теперь готовы к обработке.

События делятся на два типа; тип события задается в момент инициализации события:

- *Событие уведомления* (ручной сброс) — когда это событие устанавливается, оно освобождает любое количество ожидающих потоков, и состояние события остается установленным до того, как будет сброшено явно.
- *Событие синхронизации* (автоматический сброс) — когда это событие устанавливается, оно освобождает максимум один поток (сколько бы потоков ни ожидало события). После освобождения событие возвращается в сброшенное состояние автоматически.

Событие создается выделением памяти для структуры `KEVENT` в невыгружаемом пуле, после чего оно инициализируется вызовом функции `KeInitializeEvent` с указанием типа события (`NotificationEvent` или `SynchronizationEvent`) и исходного состояния события (установленное или сброшенное). Ожидание события выполняется обычным образом с использованием функций `KeWait`*Xxx*.

Вызов `KeSetEvent` переводит событие в установленное состояние, тогда как вызов `KeResetEvent` или `KeClearEvent` сбрасывает его (последняя функция работает чуть быстрее, так как она не возвращается к предыдущему состоянию события).

Ресурс исполнительной системы

Для решения классической задачи синхронизации — обращения к общему ресурсу из нескольких потоков — обычно использовались мьютексы или быстрые мьютексы. Такое решение работает, но мьютексы работают по пессимистической схеме; это означает, что они разрешают обращаться к общему ресурсу только одному потоку. Это может быть неэффективно в тех ситуациях, когда другие потоки обращаются к общему ресурсу только для чтения.

В тех случаях, когда изменение данных (запись) можно отличить от простого получения данных (чтения), появляется возможность оптимизации. Поток, требующий обращения к общему ресурсу, может декларировать свои намерения — чтение или запись. Если поток декларирует чтение, то другие потоки, декларирующие чтение, могут выполнять чтение конкурентно с повышением быстродействия. Это особенно полезно в тех случаях, когда общие данные изменяются медленно, то есть запись выполняется заметно реже чтения.

> Мьютексы по своей природе препятствуют конкурентности, потому что они ограничивают выполнение одним потоком в любой момент времени. В результате мьютексы всегда достигают своей цели за счет потенциального прироста быстродействия, который может быть получен при параллельном выполнении.

Ядро предоставляет еще один примитив синхронизации, предназначенный именно для подобных сценариев — «один записывает, несколько читают». Этот объект называется *ресурсом исполнительной системы* (executive resource) — еще один специальный объект, не являющийся объектом диспетчеризации.

Инициализация ресурса исполнительной системы выполняется созданием структуры `ERESOURCE` в невыгружаемом пуле и последующим вызовом `ExInitializeResourceLite`. После инициализации потоки могут захватывать ресурс либо в монопольном режиме (для записи) вызовом `ExAcquireResourceExclusiveLite`, либо в совместном режиме вызовом `ExAcquireResourceSharedLite`. После выполнения своей работы поток освобождает исполнительный ресурс вызовом `ExReleaseResourceLite` (неважно, был ли он захвачен в монопольном режиме или нет). Главное требование для использования функций захвата и освобождения — блокировка обычных APC-вызовов ядра. Это может быть сделано вызовом `KeEnterCtriticalRegion` непосредственно перед захватом и последующим вызовом `KeLeaveCriticalRegion`

после вызова функции освобождения. Следующий фрагмент кода показывает, как это делается:

```
ERESOURCE resource;

void WriteData() {
    KeEnterCriticalRegion();
    ExAcquireResourceExclusiveLite(&resource, TRUE); // Ожидание до захвата

    // Запись данных

    ExReleaseResourceLite(&resource);
    KeLeaveCriticalRegion();
}
```

Так как подобные вызовы так часто встречаются при работе с ресурсами исполнительной системы, определены специальные функции, выполняющие обе операции за один вызов:

```
void WriteData() {
    ExEnterCriticalRegionAndAcquireResourceExclusive(&resource);

    // Запись данных

    ExReleaseResourceAndLeaveCriticalRegion(&resource);
}
```

Аналогичная функция `ExEnterCriticalRegionAndAcquireResourceShared` существует для совместного захвата.

> Также существуют другие функции для работы с ресурсами исполнительной системы в некоторых специфических ситуациях. За дополнительной информацией обращайтесь к документации WDK.

Создайте соответствующие RAII-обертки C++ для ресурсов исполнительной системы.

Синхронизация при высоких уровнях IRQL

При обсуждении синхронизации до сих пор рассматривались потоки, выполняющие ожидание по объектам разных типов. Тем не менее в некоторых ситуациях потоки ожидать не могут, а именно когда IRQL процессора находится на уровне `DISPATCH_LEVEL` (2) и выше. В этом разделе рассматриваются такие ситуации и возможные решения для них.

Рассмотрим пример: драйвер устанавливает таймер при помощи функции `KeSetTimer` и использует DPC-вызов для выполнения кода при срабатывании

таймера. В то же время в драйвере могут выполняться другие функции (такие, как IRP_MJ_DEVICE_CONTROL), выполняемые при IRQL 0. Если обеим функциям необходимо обращаться к общему ресурсу (например, к связному списку), они должны синхронизировать доступ, чтобы предотвратить повреждение данных.

Проблема в том, что DPC-вызовы не могут вызывать KeWaitForSingleObject или любые другие функции ожидания — вызов любых из них окажется фатальным. Как же синхронизировать доступ для этих функций?

Однопроцессорная система — относительно простой случай. В этом случае при обращении к общему ресурсу функции с низким IRQL достаточно поднять IRQL до DISPATCH_LEVEL, а затем обращаться к ресурсу. В это время DPC не может вмешаться в выполнение кода, потому что уровень IRQL процессора уже равен 2. Завершив работу с общим ресурсом, он может снова понизить IRQL до нуля, чтобы DPC-вызов мог выполниться. Таким образом предотвращается одновременное выполнение этих функций. Схема происходящего показана на рис. 6.10.

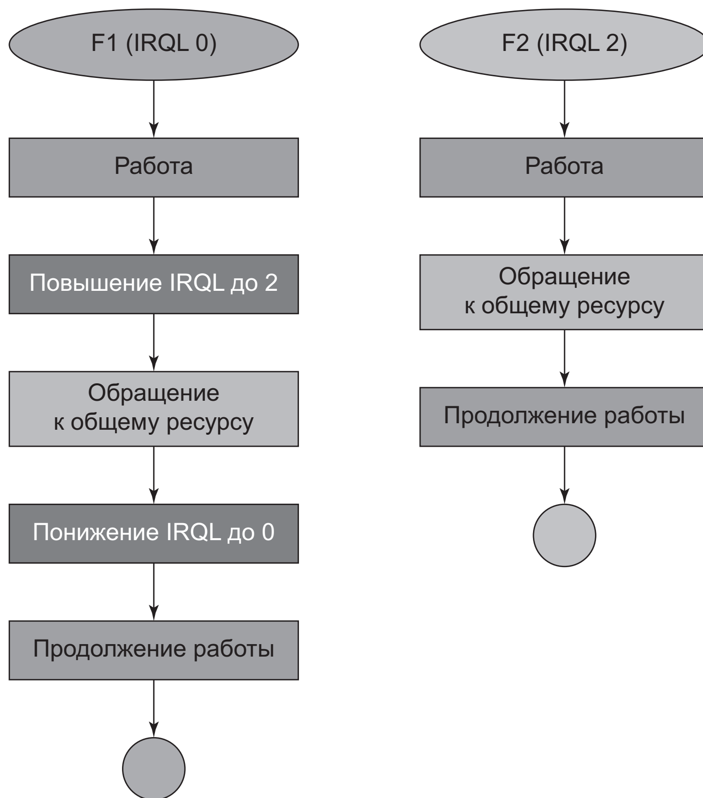


Рис. 6.10. Синхронизация с изменением IRQL

В стандартных системах с несколькими процессорами этого метода синхронизации оказывается недостаточно, потому что IRQL является свойством конкретного процессора, а не общесистемным свойством. Если IRQL одного процессора повышается до 2 и должен быть выполнен DPC-вызов, он может вмешаться в работу другого процессора, уровень IRQL которого может быть равен 0. В этом случае может оказаться, что обе функции будут выполняться одновременно и обратятся к общему ресурсу, что приведет к гонке по данным.

Как решить эту проблему? Необходимо что-то вроде мьютекса, но с возможностью синхронизации между процессорами — не потоками. Это объясняется тем, что если уровень IRQL процессора равен 2 и выше, сам поток теряет смысл, потому что планировщик не может работать с этим процессором. И такой объект действительно существует!

Спин-блокировка

Спин-блокировка (spin lock) представляется простым битом в памяти, который обеспечивает атомарность операций проверки и изменений через API. Когда процессор пытается захватить спин-блокировку, которая в настоящий момент не свободна, процессор продолжает находиться в активном ожидании ее освобождения другим процессором (напомним: перевод потока в состояние ожидания не может осуществляться на уровне `IRQL DISPATCH_LEVEL` и выше).

В сценарии, описанном в предыдущем разделе, необходимо создать и инициализировать спин-блокировку. Каждая функция, которой требуется доступ к общим данным, должна поднять уровень IRQL до 2 (если он находится на более низком уровне), захватить спин-блокировку, поработать с общими данными и напоследок освободить спин-блокировку и снова понизить IRQL (если актуально; не для DPC-вызовов). Соответствующая цепочка событий изображена на рис. 6.11.

Чтобы создать спин-блокировку, необходимо выделить память для структуры `KSPIN_LOCK` из невыгружаемого пула и вызвать `KeInitializeSpinLock`. При этом спин-блокировка переходит в незахваченное состояние.

Захват спин-блокировки всегда выполняется за два шага: сначала IRQL повышается до нужного уровня (наивысшего уровня любой функции, пытающейся синхронизировать доступ к общему ресурсу). В приведенном примере этот уровень IRQL равен 2. После этого необходимо захватить спин-блокировку. Эти два шага объединяются при помощи соответствующей функции API. Процесс изображен на рис. 6.12.

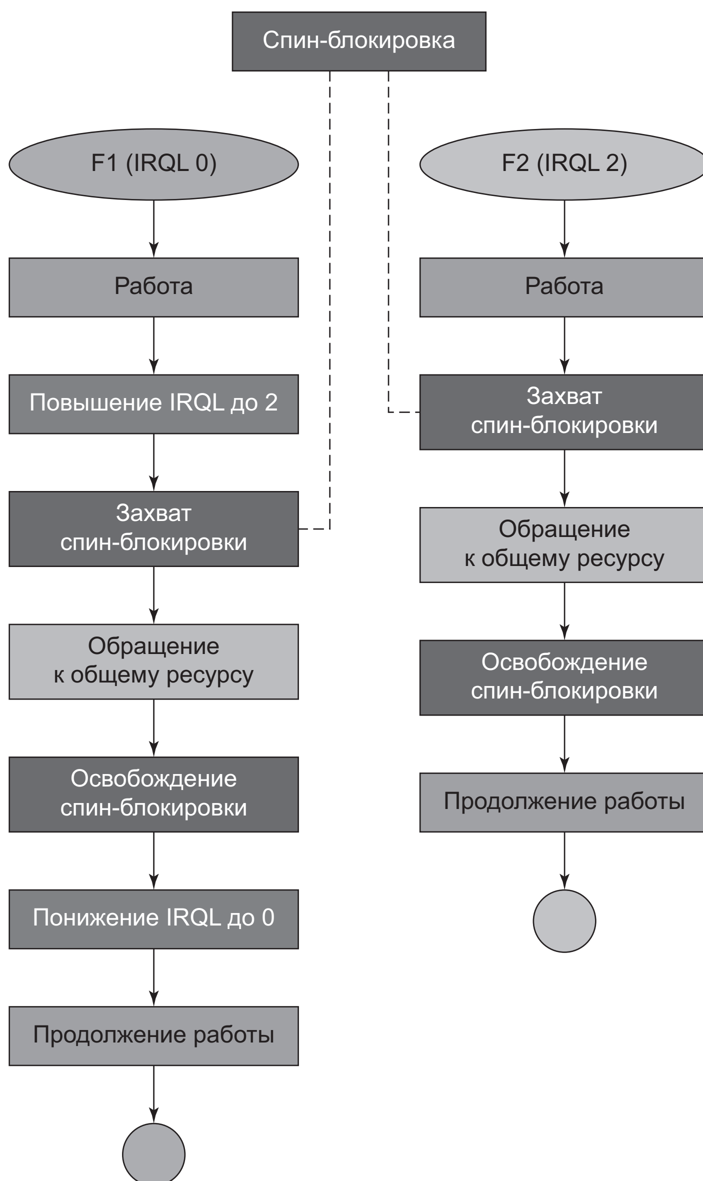


Рис. 6.11. Синхронизация при высоких IRQL с использованием спин-блокировки

Рис. 6.12. Захват спин-блокировки

Для захвата и освобождения спин-блокировки используются функции API, выполняющие два шага, показанные на рис. 6.12. В табл. 6.4 перечислены соответствующие API и связанные с ними уровни IRQL спин-блокировок, с которыми они работают.

Таблица 6.4. Функции API для работы со спин-блокировками

IRQL	**Функция захвата**	**Функция освобождения**	**Примечания**
2	`KeAcquireSpinLock`	`KeReleaseSpinLock`	
2	`KeAcquireSpinLockAtDpcLevel`	`KeReleaseSpinLockFromDpcLevel`	(1)
IRQL устройства	`KeAcquireInterruptSpinLock`	`KeReleaseInterruptSpinLock`	(2)
HIGH_LEVEL	`ExInterlockedXxx`	(нет)	(3)

Примечания к таблице 6.4:

1. Может вызываться только на уровне IRQL 2. Предоставляет оптимизацию, которая просто захватывает спин-блокировку без изменения IRQL. Канонический пример — DPC-вызов.
2. Используется для синхронизации обработчика прерывания с любой другой функцией. Драйверы оборудования с источником прерываний используются этими функциями. В аргументе передается объект прерывания — спин-блокировка является его частью.
3. Три функции для работы со связными списками на базе `LIST_ENTRY`. Эти функции используют предоставленную спин-блокировку и повышают IRQL до уровня `HIGH_LEVEL`. Из-за высокого уровня IRQL эти функции могут использоваться в любой ситуации, так как повышение IRQL всегда безопасно.

Если вы захватываете спин-блокировку, не забудьте освободить ее в той же функции. В противном случае возникает риск взаимной блокировки.

Откуда берутся спин-блокировки? В описанной ситуации драйвер должен сам создать собственную спин-блокировку для защиты параллельного доступа к своим данным из функций с высоким уровнем IRQL. Некоторые спин-блокировки существуют как часть других объектов, например объекта KINTERRUPT, используемого драйверами оборудования, которые занимаются обработкой прерываний. Другой пример — общесистемная спин-блокировка, называемая «спин-блокировкой отмены», которая захватывается ядром перед вызовом функции отмены, зарегистрированной ядром. Это единственный случай, в котором драйвер, освобождающий спин-блокировку, не захватил ее явно.

Если сразу несколько процессоров попытаются захватить одну спин-блокировку одновременно, какой процессор получит ее первым? Обычно никакого определенного порядка не существует — побеждает процессор с самыми быстрыми электронами :). Ядро предоставляет альтернативу — спин-блокировки с очередями, которые обслуживают процессоры по принципу FIFO. Они работают только с IRQL DISPATCH_LEVEL, а для работы с ними используются функции API — `KeAcquireInStackQueuedSpinLock` и `KeReleaseInStackQueuedSpinLock`. За дополнительной информацией обращайтесь к документации WDK.

Напишите обертку на C++ для спин-блокировок DISPATCH_LEVEL, работающую с классом RAII AutoLock, определенным ранее в этой главе.

Рабочие элементы

Иногда возникает необходимость в выполнении фрагмента кода в потоке, отличном от текущего. В такой ситуации можно создать поток явно и поручить ему выполнение кода. Ядро предоставляет функции, которые позволяют драйверу создать отдельный поток выполнения: `PsCreateSystemThread` и `IoCreateSystemThread` (доступна в Windows 8+). Эти функции уместны в том случае, если драйвер должен выполнять код в фоновом режиме в течение длительного времени. Тем не менее для операций, ограниченных по времени, лучше использовать предоставленный ядром пул потоков, которые будут выполнять ваш код в некотором рабочем потоке, предоставленном системой.

Функция `IoCreateSystemThread` считается предпочтительной, потому что она позволяет связать устройство или объект драйвера с потоком. Она заставляет систему ввода/вывода добавить ссылку на объект, чтобы драйвер гарантированно не был выгружен во время выполнения потока.

Поток, созданный драйвером, должен в итоге завершиться вызовом `PsTerminateSystemThread`. Эта функция никогда не возвращает управление в случае успеха.

Рабочий элемент (work item) — термин, используемый для описания функций, поставленных в очередь системного пула потоков. Драйвер может создать и инициализировать рабочий элемент указателем на функцию, которую драйвер хочет выполнить, после чего рабочий элемент ставится в очередь пула. На первый взгляд схема очень похожа на DPC-вызов; главное отличие заключается в том, что рабочие элементы всегда выполняются на уровне `IRQL PASSIVE_LEVEL`; это означает, что механизм может использоваться для выполнения операций на уровне IRQL 0 из функций, выполняемых на уровне IRQL 2. Например, если DPC-функция должна выполнить операцию, не разрешенную на уровне IRQL 2 (например, открытие файла), она может использовать рабочий элемент для выполнения этих операций.

Создание и инициализация рабочего элемента могут выполняться одним из двух способов:

- Создание и инициализация рабочего элемента функцией `IoAllocateWorkItem`. Функция возвращает указатель на непрозрачную структуру `IO_WORKITEM`. После завершения операций рабочий элемент должен быть освобожден вызовом `IoFreeWorkItem`.
- Динамическое выделение памяти для структуры `IO_WORKITEM` с размером, предоставленным функцией `IoSizeofWorkItem`. После этого вызывается

функция `IoInitializeWorkItem`. После завершений операций с рабочим элементом вызывается функция `IoUninitializeWorkItem`.

Эти функции получают объект устройства, поэтому проследите за тем, чтобы драйвер не был выгружен , пока рабочий элемент находится в очереди или выполняется.

Также существует другой набор функций API для рабочих элементов. Имена всех этих функций начинаются с Ex: например, `ExQueueWorkItem`. Эти функции не связывают рабочий элемент с чем-либо в драйвере, поэтому теоретически драйвер может быть выгружен в то время, пока рабочий элемент все еще продолжает существовать. Всегда отдавайте предпочтение функциям с префиксом Io.

Для постановки рабочего элемента в очередь вызывается функция `IoQueueWorkItem`. Определение функции выглядит так:

```
viud IoQueueWorkItem(
    _Inout_ PIO_WORKITEM IoWorkItem, // Рабочий элемент
    _In_ PIO_WORKITEM_ROUTINE WorkerRoutine, // Вызываемая функция
    _In_ WORK_QUEUE_TYPE QueueType, // Тип очереди
    _In_opt_ PVOID Context); // Значение, определяемое драйвером
```

Функция обратного вызова, которую драйвер должен предоставить, имеет следующий прототип:

```
IO_WORKITEM_ROUTINE WorkItem;

void WorkItem(
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_opt_ PVOID Context);
```

Системный пул потоков имеет несколько очередей в зависимости от приоритетов потоков, обслуживающих эти рабочие элементы. Определены следующие уровни:

```
typedef enum _WORK_QUEUE_TYPE {
    CriticalWorkQueue,          // Приоритет 13
    DelayedWorkQueue,           // Приоритет 12
    HyperCriticalWorkQueue,     // Приоритет 15
    NormalWorkQueue,            // Приоритет 8
    BackgroundWorkQueue,        // Приоритет 7
    RealTimeWorkQueue,          // Приоритет 18
    SuperCriticalWorkQueue,     // Приоритет 14
    MaximumWorkQueue,
    CustomPriorityWorkQueue = 32
} WORK_QUEUE_TYPE;
```

В документации указано, что должен использоваться тип `DelayedWorkQueue`, но в действительности можно использовать любой поддерживаемый уровень.

Существует другая функция, которая может использоваться для постановки в очередь рабочего элемента: `IoQueueWorkItemEx`. Эта функция использует другую функцию обратного вызова с дополнительным параметром, в котором хранится сам рабочий элемент. Например, это может быть полезно, если функция рабочего элемента должна освободить рабочий элемент перед выходом.

Итоги

В этой главе рассматриваются различные механизмы режима ядра, которые должен знать разработчик драйверов. В следующей главе более подробно рассматриваются пакеты запросов ввода/вывода (IRP).

Глава 7

Пакеты запросов ввода/вывода (IRP)

После того как типичный драйвер завершит свою инициализацию в `DriverEntry`, он переходит к своей основной задаче — обработке запросов. Запросы упаковываются в полудокументированную структуру пакета запроса ввода/вывода (IRP, I/O Request Packet).

В этой главе более подробно рассматриваются пакеты IRP и обработка стандартных типов IRP в драйверах.

В этой главе:

- Знакомство с IRP
- Узлы устройств
- IRP и стеки ввода/вывода
- Функции диспетчеризации
- Обращение к пользовательским буферам
- Всё вместе: драйвер Zero

Знакомство с IRP

Пакет IRP — структура, память для которой выделяется из невыгружаемого пула. Как правило, память выделяется одним из диспетчеров исполнительной системы (диспетчер ввода/вывода, диспетчер Plug & Play, диспетчер электропитания), но она также может выделяться драйвером, например, для передачи запроса другому драйверу. Сторона, ответственная за выделение памяти для IRP, также отвечает за ее освобождение.

Память IRP никогда не выделяется сама по себе. К ней всегда прилагается одна или несколько структур позиций стека ввода/вывода (`IO_STACK_LOCATION`). Собственно, при выделении памяти для IRP вызывающая сторона должна указать, сколько позиций стека ввода/вывода должно быть создано с IRP. Эти позиции стека ввода/вывода следуют в памяти непосредственно за IRP. Количество позиций стека ввода/вывода равно количеству объектов устройств в стеке устройств (стеки устройств рассматриваются в следующем разделе). Когда драйвер получает пакет IRP, он также получает указатель на саму структуру IRP, зная, что за ней следует набор позиций стека ввода/вывода, один из которых предназначен для использования драйвером. Чтобы получить правильную позицию стека ввода/вывода, этот драйвер вызывает `IoGetCurrentIrpStackLocation` (на самом деле это макрос). На рис. 7.1 изображено концептуальное представление пакетов IRP и связанных с ним позиций стека ввода/вывода.

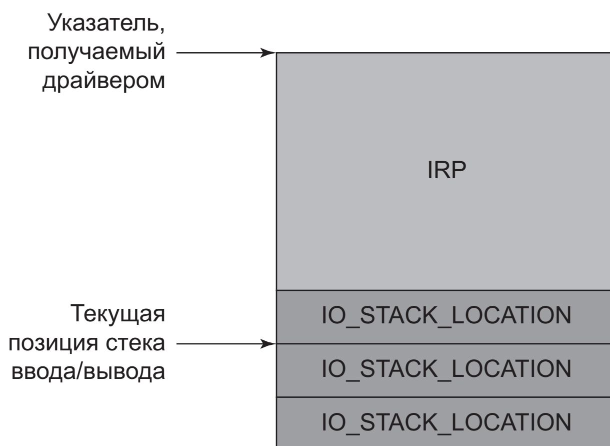


Рис. 7.1. Пакет IRP и его позиции стека ввода/вывода

Параметры запроса каким-то образом «распределены» между основной структурой IRP и текущей структурой `IO_STACK_LOCATION`.

Узлы устройств

В системе ввода/вывода в Windows центральное место занимают устройства, а не драйверы. У этого факта есть ряд последствий:

- Объектам устройств могут присваиваться имена, и для объектов устройств могут открываться дескрипторы. Функция `CreateFile` не может получать имя драйвера в аргументе.

- Windows поддерживает иерархическую структуру устройств — одно устройство может располагаться поверх другого. Это означает, что любой запрос, предназначенный для устройства более низкого уровня, сначала пройдет через верхний уровень. Иерархическая структура чаще всего встречается для физических устройств, но она также работает с любыми типами устройств.

На рис. 7.2 представлен пример нескольких уровней устройств, расположенных поверх друг друга. Такая структура называется *стеком устройств*; иногда также встречается термин «узел устройства». На схеме изображены шесть уровней, или шесть устройств. Каждое устройство в действительности представлено структурой `DEVICE_OBJECT`, созданной вызовом стандартной функции `IoCreateDevice`.

Рис. 7.2. Иерархия устройств

Имена объектов устройств, составляющих узел устройства (devnode), задаются в соответствии с их ролью в узле устройства. Эти роли актуальны для узлов устройств на основе физического оборудования.

> Все объекты устройств на рис. 7.2 представляют собой обычные структуры DEVICE_OBJECT, создаваемые разными драйверами, ответственными за этот уровень. Впрочем, в общем случае такая разновидность узлов устройств не обязательно связывается с драйверами физического оборудования.

Краткая сводка меток на рис. 7.2:

- *PDO* (Physical Device Object) — несмотря на название, в объекте нет ничего «физического» («physical»). Этот объект устройства создается драйвером шины, то есть драйвером, обслуживающим конкретную шину (PCI, USB и т. д.). Объект устройства отражает тот факт, что в слоте шины присутствует некоторое устройство.
- *FDO* (Functional Device Object) — этот объект устройства создается «настоящим» драйвером, то есть драйвером, который обычно предоставляется поставщиком оборудования и во всех подробностях знает все тонкости реализации устройства.
- *FiDO* (Filter Device Object) — необязательные устройства-фильтры, создаваемые драйверами-фильтрами.

Диспетчер Plug & Play (P&P) в данном случае отвечает за загрузку соответствующих драйверов, начиная с нижнего уровня. Например, представьте, что узел устройства на рис. 7.2 представляет набор драйверов, управляющих сетевым адаптером с интерфейсом PCI. Последовательность событий, приводящих к созданию этого узла устройства, может быть представлена следующим образом:

1. Драйвер шины PCI (pci.sys) распознает факт присутствия чего-то в конкретном слоте. Он создает структуру PDO (`IoCreateDevice`) для представления этого факта. Драйвер шины понятия не имеет, что это — сетевой адаптер, видеокарта или что-нибудь еще; он знает только то, что устройство присутствует, и может получить базовую информацию от своего контроллера (например, узнать идентификатор поставщика и идентификатор устройства).
2. Драйвер шины PCI уведомляет диспетчера P&P о том, что на шине обнаружены изменения.
3. Диспетчер P&P запрашивает список объектов PDO, находящихся под управлением драйвера шины. Он получает список PDO, в который входит новый объект PDO.
4. Теперь диспетчер P&P должен найти и загрузить подходящий драйвер для нового PDO. Он выдает запрос к драйверу шины, чтобы запросить полный идентификатор оборудования.
5. Зная идентификатор оборудования, диспетчер P&P обращается к реестру `HKLM\System\CurrentControlSet\Enum\PCI\`*(HardwareID)*. Если драйвер загружался ранее, он будет зарегистрирован здесь, и диспетчер P&P его загрузит. На рис. 7.3 изображен пример идентификатора оборудования (драйвер экрана от NVIDIA) в реестре.

6. Драйвер загружает и создает FDO (еще один вызов `IoCreateDevice`), но добавляет дополнительный вызов `IoAttachDeviceToDeviceStack`, присоединяя себя к предыдущему уровню (обычно PDO).

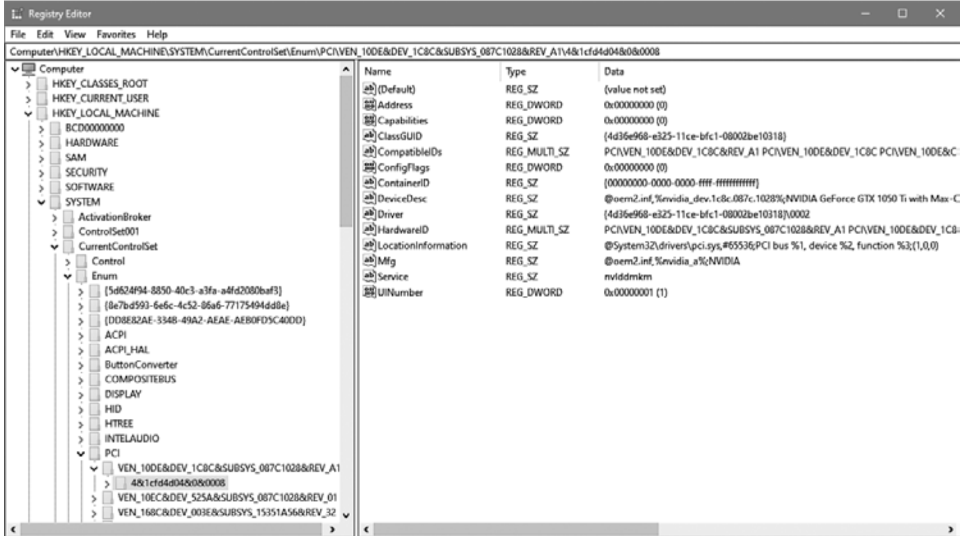


Рис. 7.3. Информация об идентификаторах оборудования

Значение Service на рис. 7.3 косвенно ссылается на фактический драйвер из раздела `HKLMSystemCurrentControlSetServices{ServiceName}`, где должны регистрироваться все драйверы.

Также загружаются объекты устройств-фильтров, если они были правильно зарегистрированы в реестре. Фильтры нижних уровней (ниже FDO) загружаются по порядку, начиная с нижних. Каждый загруженный драйвер-фильтр создает собственный объект устройства и присоединяет его над предыдущим уровнем. Верхние уровни работают аналогичным образом, но они загружаются после FDO. Все это означает, что с рабочими узлами устройств P&P существуют по крайней мере два уровня — PDO и FDO, но их может быть и больше при участии фильтров. Основы разработки фильтров для драйверов оборудования рассматриваются в главе 11.

> Полное обсуждение Plug & Play и подробности построения узлов устройств выходят за рамки книги. Предыдущее описание неполно, в нем пропущены некоторые подробности, но оно дает общее представление о происходящем. Каждый узел устройства строится снизу вверх независимо от того, связан он с оборудованием или нет.

Поиск нижних фильтров осуществляется в двух местах: в разделе идентификаторов оборудования на рис. 7.3 и в соответствующем классе, основанном на значении `ClassGuid` из раздела `HKLM\System\CurrentControlSet\ControlClasses`. Параметр называется `LowerFilters` и представляет собой многострочное значение с именами служб. Поиск верхних фильтров происходит аналогичным образом, но для параметра с именем `UpperFilters`. На рис. 7.4 показано содержимое реестра для класса `DiskDrive`, у которого имеется как нижний, так и верхний фильтр.

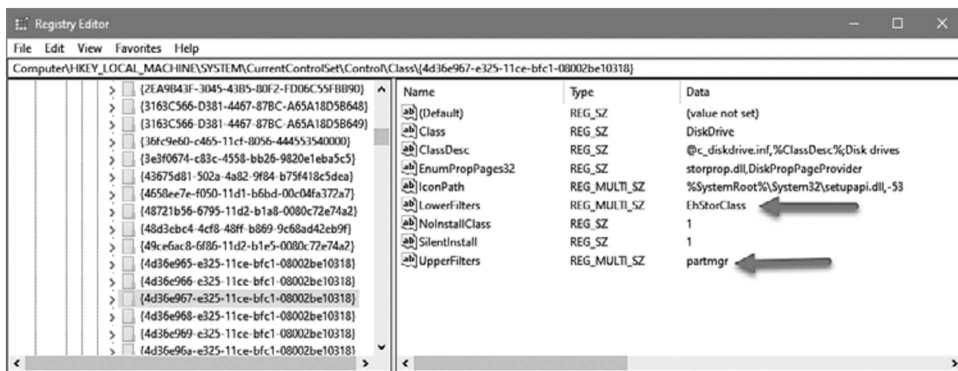


Рис. 7.4. Раздел класса DiskDrive

Последовательность действий при работе с IRP

На рис. 7.2 показан пример узла устройства (независимо от того, связан он с конкретным устройством или нет). Пакет IRP создается одним из диспетчеров исполнительной системы — для большинства наших драйверов это диспетчер ввода/вывода.

Диспетчер создает IRP с сопутствующими структурами `IO_STACK_LOCATION` — на рис. 7.2 их шесть. Диспетчер инициализирует главную структуру IRP и только первую позицию стека ввода/вывода. Затем указатель на IRP передается самому верхнему уровню.

Драйвер получает пакет IRP в соответствующей функции диспетчеризации. Например, если пакет предназначен для чтения, драйвер будет вызван по индексу `IRP_MJ_READ` массива `MajorFunction` из объекта драйвера.

На этот момент у драйвера есть несколько вариантов обработки IRP:

1. Передать запрос вниз — если устройство драйвера не является последним устройством в узле, драйвер может передать запрос далее, если он не представляет интереса для драйвера. Обычно так поступают фильтрующие

драйверы при получении запроса, который для них неактуален: чтобы не повредить функциональность устройства (так как запрос предназначен для устройства более низкого уровня), драйвер передает его вниз. Это может быть сделано при помощи двух функций:

- Вызовите `IoSkipCurrentIrpStackLocation`, чтобы следующее устройство в цепочке гарантированно увидело ту же информацию, переданную этому устройству.
- Вызовите `IoCallDriver` с передачей нижнего объекта устройства (который был получен драйвером в момент вызова `IoAttachDeviceToDeviceStack`) и IRP.

> Прежде чем передавать запрос вниз, драйвер должен подготовить следующую позицию стека ввода/вывода с правильной информацией. Так как диспетчер ввода/вывода инициализирует только первую позицию стека ввода/вывода, каждый драйвер сам несет ответственность за то, чтобы инициализировать следующую позицию. Одно из возможных решений — вызвать `IoCopyIrpStackLocationToNext` перед вызовом `IoCallDriver`. Такое решение работает, но оно неэффективно, если драйвер просто хочет, чтобы нижний уровень увидел ту же информацию. Вызов `IoSkipCurrentIrpStackLocation` — оптимизация, которая уменьшает указатель на текущую позицию стека ввода/вывода внутри IRP, которая позднее увеличивается вызовом `IoCallDriver`, так что следующий уровень увидит ту же позицию IO_STACK_LOCATION, которую видел драйвер. Трюк с уменьшением/увеличением работает эффективнее, чем фактическое копирование.

2. Полностью обработать пакет IRP — драйвер, получающий пакет IRP, может просто обработать его без передачи вниз, с завершающим вызовом `IoCompleteRequest`. Устройства более низких уровней вообще не увидят запрос.
3. Применить комбинацию (1) и (2) — драйвер может проанализировать IRP, что-то сделать (например, зарегистрировать запрос в журнале) и передать его. Или он может внести изменения в следующую позицию стека ввода/вывода и передать запрос вниз.
4. Передать запрос вниз и получить уведомление, когда запрос будет завершен устройством более низкого уровня: любой уровень (кроме самого нижнего) может назначить функцию завершения ввода/вывода, вызвав `IoSetCompletionRoutine` перед тем, как передавать запрос вниз. Когда запрос будет завершен на одном из более низких уровней, будет вызвана функция завершения драйвера.
5. Начать асинхронную обработку IRP — драйвер может захотеть обработать запрос, но если обработка занимает много времени (типично для

драйверов оборудования, но также может встречаться для программных драйверов), драйвер может пометить IRP как незавершенный вызовом `IoMarkIrpPending`, после чего вернуть `STATUS_PENDING` из функции диспетчеризации. Рано или поздно он должен будет завершить IRP.

Когда какой-то уровень вызовет `IoCompleteRequest`, пакет IRP разворачивается и начинает «подниматься» к источнику IRP (как правило, это один из диспетчеров). Если функции завершения были зарегистрированы, они будут вызваны в порядке, обратном порядку регистрации, то есть от низа к верху.

> В большинстве драйверов, приведенных в книге, иерархия не учитывается, так как драйвер с большой вероятностью будет частью одноуровневого узла устройства. В этом случае драйвер обрабатывает запрос «на месте».
>
> Другие аспекты обработки IRP рассматриваются при обсуждении драйверов-фильтров в главе 11.

IRP и позиция стека ввода/вывода

На рис. 7.5 представлены некоторые важные поля IRP.

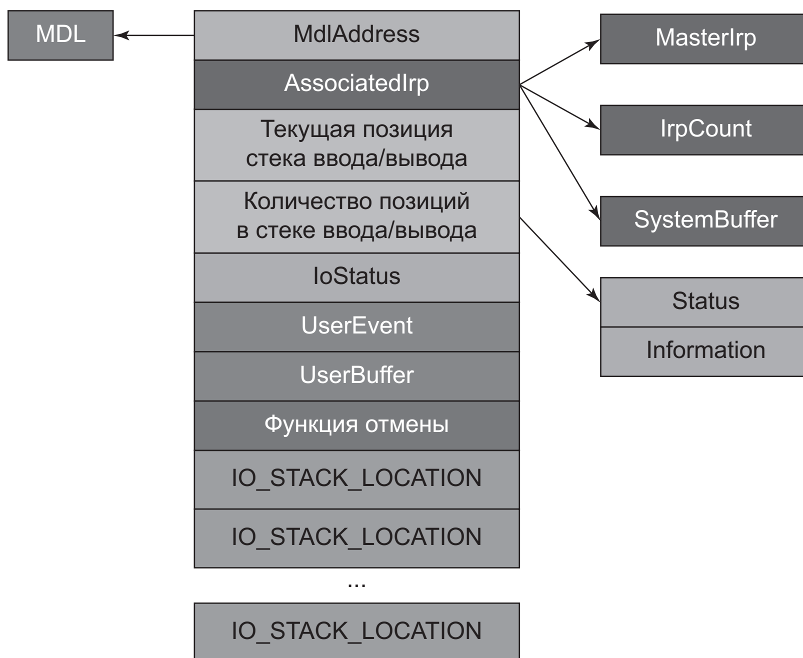


Рис. 7.5. Важные поля структуры IRP

Краткое описание полей структуры:

- IoStatus — содержит статус (NT_STATUS) пакета IRP и поле Information. Поле Information является полиморфным: оно объявлено с типом ULONG_PTR (32- или 64-разрядное целое число), но его смысл зависит от типа IRP. Например, для IRP чтения и записи оно интерпретируется как количество байтов, передаваемых в ходе операции.
- UserBuffer — содержит указатель на низкоуровневый буфер для IRP. Например, IRP чтения и записи хранят в этом поле указатель на пользовательский буфер, а у IRP DeviceIoConttrol хранится указатель на выходной буфер, предоставляемый в запросе.
- UserEvent — указатель на объект события (KEVENT), предоставленный клиентом, если вызов является асинхронным, и такое событие было предоставлено. Из пользовательского режима это событие может быть предоставлено (структурой HANDLE) в структуре OVELAPPED, обязательной для асинхронного вызова операций ввода/вывода.
- AssociatedIrp — объединение состоит из трех составляющих, из которых может быть действительна только одна (максимум):
  - MasterIrp — указатель на «главный» пакет IRP, если текущий пакет IRP является ассоциированным. Диспетчер ввода/вывода поддерживает возможность создания «главного» пакета IRP, с которым может быть связано несколько «ассоциированных» пакетов IRP. После того как завершатся все ассоциированные IRP, главный IRP завершится автоматически. Поле MasterIrp актуально для ассоциированных IRP — в нем хранится указатель на главный IRP.
  - IrpCount — для главного пакета IRP это поле обозначает количество ассоциированных IRP.
  - SystemBuffer — (используется чаще всего) указатель на буфер, выделенный системой из невыгружаемого пула и используемый для буферизованных операций ввода/вывода. За подробностями обращайтесь к разделу «Буферизованный ввод/вывод» этой главы.

  Главные и ассоциированные IRP встречаются относительно редко. В книге этот механизм использоваться не будет.

- *Функция отмены* — указатель на функцию отмены, которая будет вызвана (если значение отлично от NULL) по запросу на отмену операции, например, функциями пользовательского режима CancelIo и CancelIoEx. Программным драйверам функции отмены обычно не нужны, поэтому в книге они не используются.

- `MdlAddress` — указатель на необязательный список дескрипторов памяти MDL (Memory Descriptor List). MDL — структура данных ядра, которая умеет описывать буферы в памяти. `MdlAddress` используется прежде всего с прямым вводом/выводом (см. раздел «Прямой ввод/вывод» этой главы).

Каждый пакет IRP сопровождается одной или несколькими структурами `IO_STACK_LOCATION`. На рис. 7.6 представлены важные поля `IO_STACK_LOCATION`.

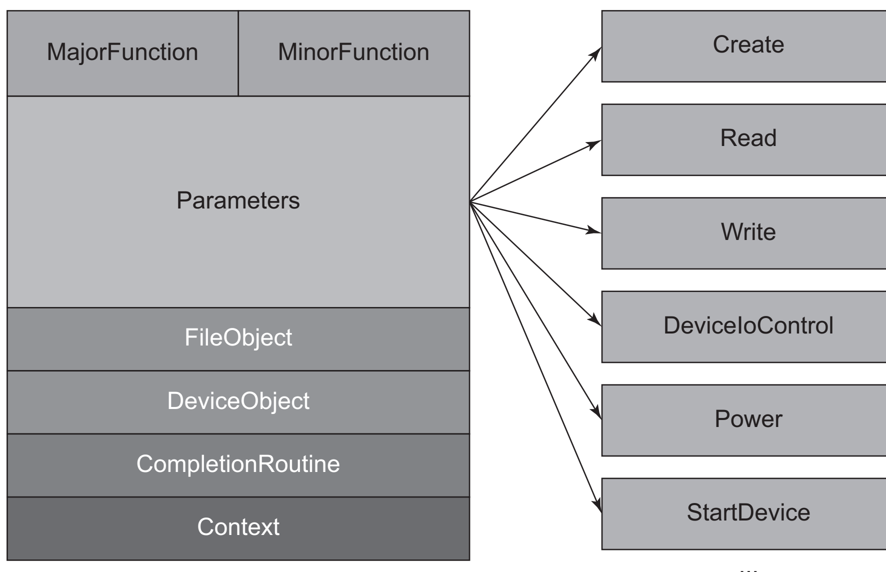


Рис. 7.6. Важные поля структуры IO_STACK_LOCATION

Краткое описание полей на рис. 7.6:

- `MajorFunction` — первичная функция IRP (`IRP_MJ_CREATE`, `IRP_MJ_READ` и т. д.). Поле иногда используется, если драйвер связывает несколько кодов первичных функций с одним обработчиком. В этой функции драйвер может идентифицировать нужную функцию при помощи этого поля.
- `MinorFunction` — некоторые типы IRP имеют дополнительные функции: `IRP_MJ_PNP`, `IRP_MJ_POWER` и `IRP_MJ_SYSTEM_CONTROL` (WMI). Типичный код таких обработчиков содержит команду `switch` с выбором по значению `MinorFunction`. В книге эти виды IRP использоваться не будут, кроме драйверов-фильтров для физических устройств, которые будут более подробно рассмотрены в главе 11.

- FileObject — структура FILE_OBJECT, связанная с IRP. В большинстве случаев она не используется, но доступна для тех функций диспетчеризации, которым может понадобиться.
- DeviceObject — объект устройства, связанный с IRP. Функции диспетчеризации получают указатель на этот объект, поэтому обычно обращаться к этому полю не требуется.
- CompletionRoutine — функция завершения, назначенная для предыдущего (верхнего) уровня вызовом IoSetCompletionRoutine.
- Context — аргумент, передаваемый функции завершения (если она есть).
- Parameters — огромное объединение, содержащее несколько структур, каждая из которых актуальна для конкретной операции. Например, в операции чтения (IRP_MJ_READ) поле Parameters.Read содержит структуру, из которой можно получить дополнительную информацию об операции чтения.

Текущая позиция стека ввода/вывода, полученная вызовом IoGetCurrentIrpStackLocation, содержит большинство параметров запроса в объединении Parameters. Задача драйвера — обратиться к правильной структуре, как уже было показано в главе 4 (и будет снова показано в этой и последующих главах).

Просмотр информации об IRP

В процессе отладки или анализа дампов ядра есть пара команд, которые могут пригодиться при поиске или анализе IRP.

- Команда !irpfind может использоваться для поиска пакетов IRP — либо всех, либо отвечающих определенному критерию. Без аргументов команда !irpfind ищет в невыгружаемом пуле(-ах) все IRP. В документации отладчика можно узнать, как задавать конкретные критерии для ограничения поиска. Пример вывода при поиске всех IRP:

```
lkd> !irpfind
Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool

Scanning large pool allocation table for tag 0x3f707249 (Irp?)
                                             (ffffbf0a87610000 : f\
fffbf0a87910000)

  Irp           [ Thread ]          irpStack: (Mj,Mn)   DevObj          [Driver] \
MDL Process
```

```
ffffbf0aa795ca30 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a9a8ef010 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a8e68ea20 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a90deb710 [ffffbf0a808a1080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a99d1da90 [0000000000000000] Irp is complete (CurrentLocation 10 > \
StackCount9)
ffffbf0a74cec940 [0000000000000000] Irp is complete (CurrentLocation 8 > \
StackCount7)
ffffbf0aa0640a20 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a89acf4e0 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]
ffffbf0a89acfa50 [ffffbf0a7fcde080] irpStack: ( c, 2) ffffbf0a74d20050 [ \
FileSystem\Ntfs]

(...)
```

Для конкретного IRP команда `!irp` анализирует пакет и выводит удобную сводку его данных. Как обычно, команда `dt` может использоваться с типом `_IRP` для просмотра всей структуры IRP.

Пример просмотра структуры IRP командой `!irp`:

```
kd> !irp ffffbf0a8bbada20
Irp is active with 13 stacks 12 is current (= 0xffffbf0a8bbade08)
 No Mdl: No System Buffer: Thread ffffbf0a7fcde080: Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [N/A(0), N/A(0)]
    0  0 00000000 00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
    0  0 00000000 00000000 00000000-00000000

(...)

    Args: 00000000 00000000 00000000 00000000
 [N/A(0), N/A(0)]
    0  0 00000000  00000000 00000000-00000000

    Args: 00000000 00000000 00000000 00000000
>[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
    0 e1 ffffbf0a74d20050 ffffbf0a7f52f790 fffff8015c0b50a0-ffffbf0a91d99010 \
Success
 Error Cancel pending
      \FileSystem\Ntfs
       Args: 00004000 00000051 00000000 00000000
 [IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
```

```
0  0 ffffbf0a60e83dc0 ffffbf0a7f52f790 00000000-00000000
   \FileSystem\FltMgr
Args: 00004000 00000051 00000000 00000000
```

Команды `!irp` выводят позиции стека ввода/вывода и информацию, хранящуюся в них.

Функции диспетчеризации

Как упоминалось в главе 4, одним из важных аспектов `DriverEntry` является настройка функций диспетчеризации. Эти функции связаны с кодами первичных функций. Поле `majorFunction` в `DRIVER_OBJECT` содержит массив указателей на функции, индексируемый по коду первичной функции.

Все функции диспетчеризации имеют одинаковый прототип; повторим его для удобства с использованием определения типа `DRIVER_DISPATCH` из WDK (несколько упрощено для ясности):

```
typedef NTSTATUS DRIVER_DISPATCH (
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp);
```

Функция диспетчеризации (выбранная на основании кода первичной функции) становится первой функцией драйвера, которая увидит запрос. Обычно она вызывается в контексте запрашивающего потока, то есть потока, который вызывал соответствующую функцию API (например, `ReadFile`) на уровне IRQL `PASSIVE_LEVEL` (0). Тем не менее может случиться, что драйвер-фильтр, находящийся выше этого устройства, отправил устройство вниз в другом контексте — это может быть поток, не связанный с исходным запрашивающим потоком и даже выполняемый на более высоком уровне IRQL (например, `DISPATCH_LEVEL` (2)).

Хорошо написанный драйвер должен быть готов к подобным ситуациям, несмотря на то что у программных драйверов «неудобный» контекст встречается редко. О том, как корректно обрабатывать подобные ситуации, рассказано в разделе «Обращение к пользовательским буферам» этой главы.

Все функции диспетчеризации выполняют определенный набор операций:

1. Проверка ошибок — обычно функция диспетчеризации начинает с проверки логических ошибок. Например, операции чтения и записи используют буферы — имеют ли эти буферы правильные размеры? Для запросов `DeviceIoControl` наряду с двумя возможными буферами присутствует код управляющей операции. Драйвер должен убедиться в том, что этот код от-

носится к числу поддерживаемых. Если будет обнаружена ошибка, IRP немедленно завершается с соответствующим статусом.

2. Обработка запроса.

Список основных функций диспетчеризации для программного драйвера:

- `IRP_MJ_CREATE` — соответствует вызову `CreateFile` из пользовательского режима или `ZwCreateFile` в режима ядра. Его первичная функция является обязательной, в противном случае клиент не сможет открыть дескриптор устройства, находящегося под управлением драйвера. Многие драйверы просто завершают IRP со статусом успеха.
- `IRP_MJ_CLOSE` — по смыслу противоположен `IRP_MJ_CREATE`. Вызывается функцией `CloseHandle` из пользовательского режима или `ZwClose` из режима ядра перед закрытием последнего дескриптора объекта файла. Многие драйверы просто успешно завершают запрос, но если в `IRP_MJ_CREATE` выполнялась какая-то содержательная работа, она должна быть отменена.
- `IRP_MJ_READ` — соответствует операции чтения, обычно вызывается из пользовательского режима функцией `ReadFile` или из режима ядра функцией `ZwReadFile`.
- `IRP_MJ_WRITE` — соответствует операции записи, обычно вызывается из пользовательского режима функцией `WriteFile` или из режима ядра функцией `ZwWriteFile`.
- `IRP_MJ_DEVICE_CONTROL` — соответствует вызову `DeviceIoControl` из пользовательского режима или `ZwDeviceIoControlFile` из режима ядра (в режиме ядра также существуют другие функции API, которые могут генерировать IRP `IRP_MJ_DEVICE_CONTROL`).
- `IRP_MJ_INTERNAL_DEVICE_CONTROL` — напоминает `IRP_MJ_DEVICE_CONTROL`, но доступен только для вызова из режима ядра.

Завершение запроса

После того как драйвер решит обработать IRP (то есть не станет передавать его вниз другому драйверу), он должен в конечном итоге завершить его. В противном случае возникнет утечка — запрашивающий поток не сможет завершиться, и как следствие, содержащий его процесс тоже продолжит существование; возникает так называемый процесс-зомби.

Завершение запроса означает вызов `IoCompleteRequest` после заполнения статуса запроса и дополнительной информации. Если завершение будет

выполнено в самой функции диспетчеризации (типичный случай для программных драйверов), функция должна вернуть тот же статус, который был помещен в IRP.

Следующий фрагмент кода демонстрирует завершение запроса в функции диспетчеризации:

```
NTSTATUS MyDispatchRoutine(PDEVICE_OBJECT, PIRP Irp) {
    //...
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = NumberOfBytesTransfered; // зависит от запроса
ре
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
```

Так как функция диспетчеризации должна вернуть тот же статус, который был помещен в IRP, появляется искушение записать последнюю команду в виде:

```
return Irp->IoStatus.Status;
```

Однако это с большой вероятностью приведет к фатальному сбою системы. Сможете догадаться почему?

После того как пакет IRP будет завершен, обращения к любым его компонентам нежелательны. Вероятно, пакет IRP уже был освобожден и вы обращаетесь к освобожденной памяти. На самом деле все может быть еще хуже, потому что на его месте может быть выделена память для другого пакета IRP (распространенная ситуация), поэтому код может вернуть статус некоторого случайного пакета IRP.

Поле `Information` должно быть равно 0 в случае ошибки (статус неудачи). Его конкретный смысл для успешной операции зависит от типа IRP.

Функция `IoCompleteRequest` получает два аргумента: сам пакет IRP и необязательное временное приращение приоритета исходного потока (потока, который инициировал запрос). В большинстве случаев для программных драйверов этим потоком будет выполняемый поток, так что приращение приоритета неактуально. Значение `IO_NO_INCREMENT` определяется как 0; таким образом, в приведенном выше фрагменте приращение не используется.

Впрочем, драйвер может решить повысить приоритет потока независимо от того, является он вызывающим потоком или нет. В этом случае приоритет потока увеличивается с заданным приращением, и поток получает возможность выполнить один квант с новым приоритетом; приоритет уменьшается на 1, поток выполняет еще один квант с пониженным приоритетом и т. д., пока приоритет не вернется к исходному уровню. Ситуация представлена на рис. 7.7.

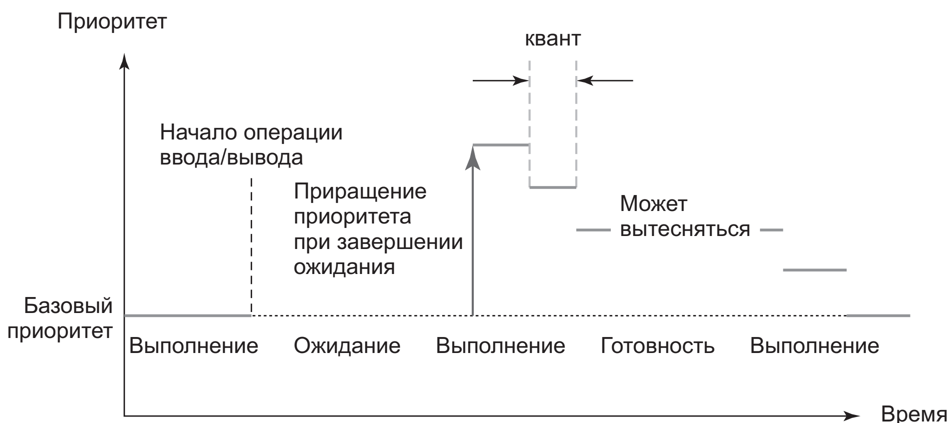


Рис. 7.7. Увеличение и постепенное снижение приоритета потока

Приоритет потока после повышения не может превысить 15. В случае превышения приоритет будет равен 15. Если приоритет исходного потока был выше 15, то повышение не имеет эффекта.

Обращение к пользовательским буферам

Заданная функция диспетчеризации первой увидит пакет IRP. Некоторые функции диспетчеризации (в первую очередь `IRP_MJ_READ`, `IRP_MJ_WRITE` и `IRP_MJ_DEVICE_CONTROL`) получают буферы, предоставленные клиентом, — в большинстве случаев из пользовательского режима. Обычно функция диспетчеризации вызывается на уровне IRQL 0 в контексте запрашивающего потока; это означает, что обращения по указателям на буферы, предоставленные пользовательским режимом, происходят тривиально: уровень IRQL равен 0, ошибки страниц обрабатываются нормально, запрос был выдан тем же потоком, поэтому указатели действительны в контексте процесса.

И все же могут возникнуть проблемы. Как было показано в главе 6, даже в этом удобном контексте (запрашивающий поток и IRQL 0) другой поток в процессе клиента может освободить переданный буфер(-ы) до того, как драйвер получит возможность прочитать их, а это приведет к нарушению прав доступа.

В главе 6 мы использовали блок `__try`/`__except` для обработки любых нарушений прав доступа, при этом клиенту возвращался признак неудачи.

В некоторых случаях даже этого оказывается недостаточно. Например, если некий код выполняется на уровне IRQL 2 (скажем, DPC-вызов, выполняемый

в результате истечения таймера), вы не можете безопасно обратиться к пользовательским буферам в таком контексте. Возникают две проблемы:

- IRQL находится на уровне 2; это означает, что обработка ошибок страниц невозможна.
- DPC-вызов выполняется в произвольном потоке, поэтому указатель сам по себе не имеет смысла в процессе, который окажется текущим на этом процессоре.

Обработка исключений в таких случаях будет работать некорректно, потому что мы обращаемся к адресу памяти, недействительному в контексте этого случайного процесса. Даже если попытка завершится успешно (потому что область памяти будет выделена в этом случайном процессе и будет находиться в ОЗУ), вы обратитесь к случайной памяти, а не к исходному буферу, который был передан с запросом.

Все это означает, что должен существовать какой-то способ обращения к буферу исходного пользователя в «неудобном» контексте. На самом деле ядро предоставляет для этой цели целых два способа: буферизованный ввод/вывод и прямой ввод/вывод. В ближайших разделах вы узнаете, как работают эти две схемы, и научитесь ими пользоваться.

Некоторые структуры данных всегда безопасны, поскольку они выделяются из невыгружаемого пула. Типичные примеры — объекты устройств (созданные функцией `IoCreateDevice`) и IRP.

Буферизованный ввод/вывод

Буферизованный ввод/вывод — простейший из двух вариантов. Чтобы получить поддержку буферизованного ввода/вывода для операций чтения и записи, необходимо установить в объекте устройства специальный флаг:

```
DeviceObject->Flags |= DO_BUFFERED_IO; // DO = Device Object
```

За информацией о буферах `IRP_MJ_DEVICE_CONTROL` обращайтесь к разделу «Пользовательские буферы для запросов IRP_MJ_DEVICE_CONTROL» этой главы.

При поступлении запроса на чтение или запись диспетчер ввода/вывода и драйвер выполняют следующие действия:

1. Диспетчер ввода/вывода выделяет в невыгружаемом пуле буфер с таким же размером, как у пользовательского буфера. Указатель на новый буфер сохраняется в поле `AssociatedIrp->SystemBuffer` пакета IRP. (Размер буфера можно узнать из поля `Parameters.Read.Length` или `Parameters.Write.Length` текущей позиции стека ввода/вывода.)

2. Для запроса на запись диспетчер ввода/вывода копирует буфер пользователя в системный буфер.
3. Только теперь вызывается функция диспетчеризации драйвера. Драйвер может использовать указатель на системный буфер напрямую без проверок, потому что буфер находится в системном пространстве (его адрес абсолютен, то есть одинаков в контексте любого процесса) на любом уровне IRQL, потому что буфер выделяется из невыгружаемого пула (а следовательно, не может быть выгружен).
4. После того как драйвер завершит IRP (`IoCompleteRequest`), диспетчер ввода/вывода (для запросов чтения) копирует системный буфер обратно в пользовательский буфер (размер копии определяется полем `IoStatus.Information` в IRP, значение которого задается драйвером).
5. Наконец, диспетчер ввода/вывода освобождает системный буфер.

Возможно, вас интересует, как диспетчер ввода/вывода копирует системный буфер в пользовательский буфер из `IoCompleteRequest`? Эта функция может быть вызвана из любого потока при IRQL <= 2. Операция выполняется постановкой в очередь специального APC-вызова ядра к потоку, запросившему операцию. Когда потоку будет предоставлен процессор для выполнения, он прежде всего выполняет этот APC-вызов, который и осуществляет непосредственное копирование.

На рис. 7.8, *а–д* изображена последовательность действий при буферизованном вводе/выводе.

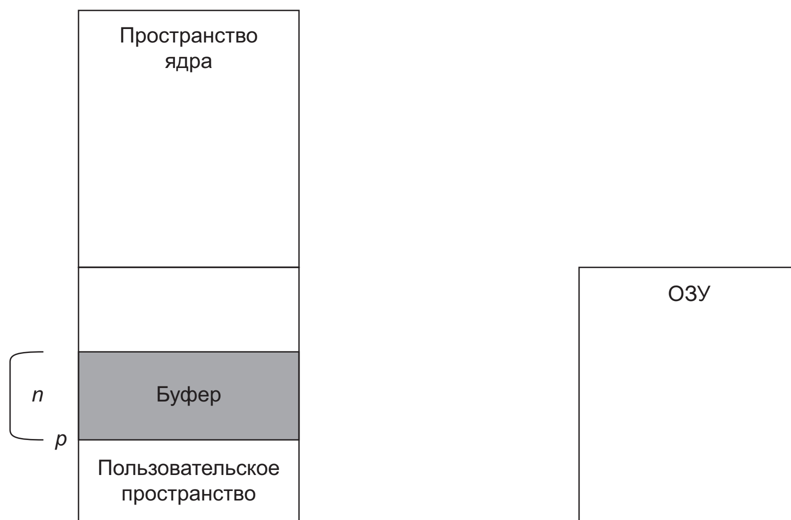


Рис. 7.8а. Буферизованный ввод/вывод: исходное состояние

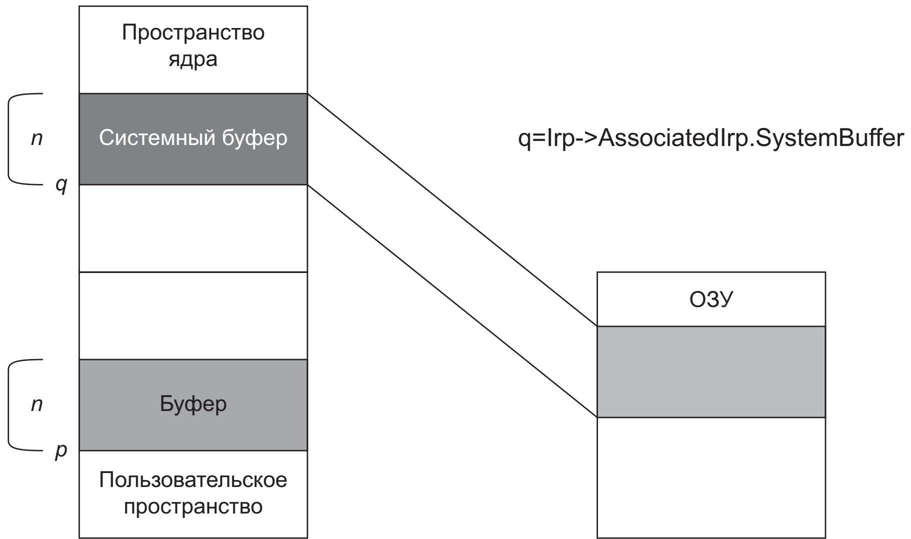


Рис. 7.8б. Буферизованный ввод/вывод: выделение системного буфера

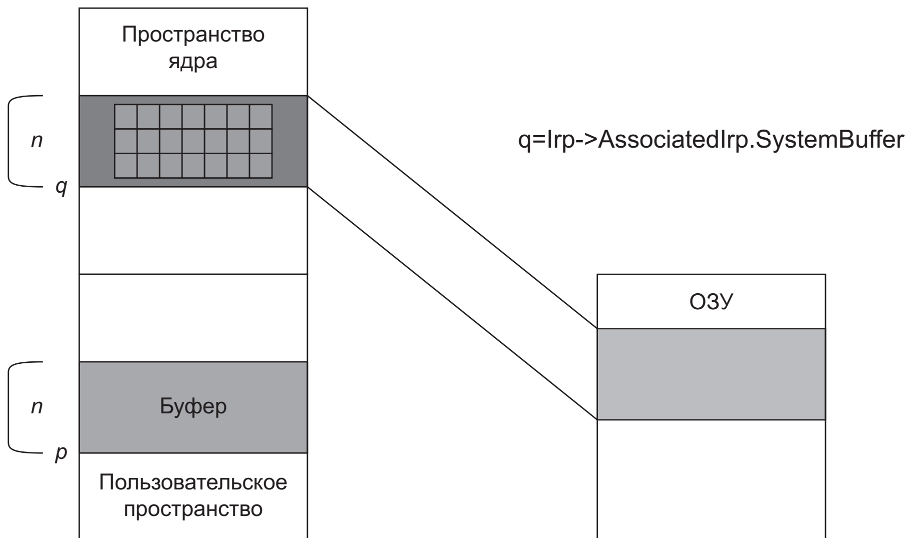


Рис. 7.8в. Буферизованный ввод/вывод: драйвер обращается к системному буферу

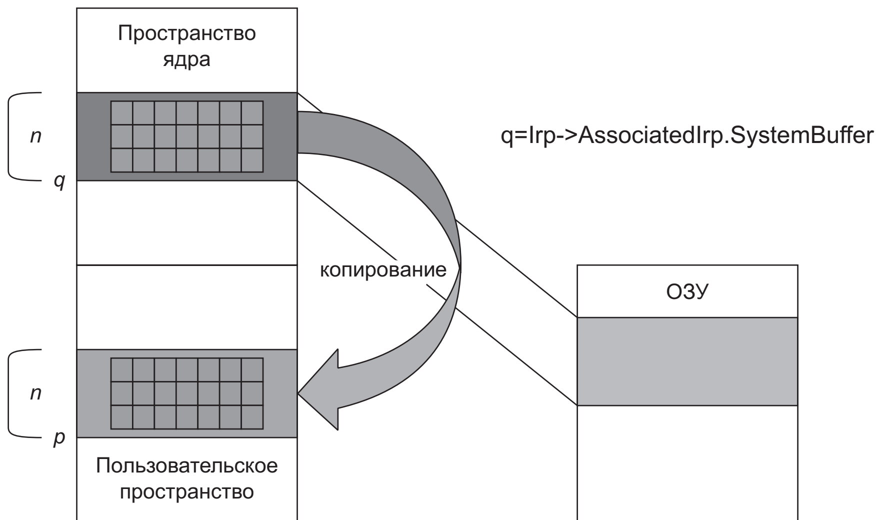


Рис. 7.8г. Буферизованный ввод/вывод: при завершении IRP диспетчер ввода/вывода копирует буфер обратно (для чтения)

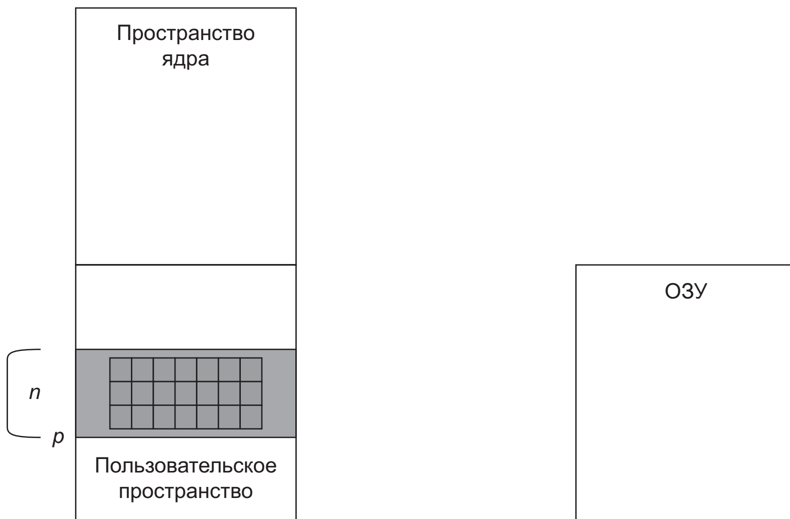


Рис. 7.8д. Буферизованный ввод/вывод: итоговое состояние — диспетчер ввода/вывода освобождает системный буфер

Буферизованный ввод/вывод обладает следующими характеристиками:

- Простота использования — просто установите флаг в объекте устройства, а все остальное будет сделано за вас автоматически диспетчером ввода/вывода.

✦ Он всегда подразумевает копирование — это означает, что его лучше использовать для небольших буферов (обычно не более одной страницы). Копирование больших буферов может обходиться слишком дорого. В таких случаях следует использовать другой вариант — прямой ввод/вывод.

Прямой ввод/вывод

Механизм прямого ввода/вывода существует для того, чтобы к пользовательскому буферу можно было обращаться на любом уровне IRQL и в любом потоке без какого-либо копирования.

Для запросов чтения и записи прямой ввод/вывод выбирается при помощи другого флага объекта устройства:

```
DeviceObject->Flags |= DO_DIRECT_IO;
```

Как и при буферизованном вводе/выводе, этот выбор влияет только на запросы чтения и записи. Запросы `DeviceIoControl` рассматриваются в следующем разделе.

Основная последовательность операций при прямом вводе/выводе:

1. Диспетчер ввода/вывода сначала убеждается в том, что пользовательский буфер действителен, после чего выводит его в физическую память.
2. Затем он блокирует буфер в памяти, чтобы буфер не мог быть выгружен до последующего уведомления. При этом решается одна из проблем обращения к буферам — ошибки страниц невозможны, поэтому обращение к буферу на любом уровне IRQL безопасно.
3. Диспетчер ввода/вывода строит MDL (Memory Descriptor List) — структуру данных, которая знает, как буфер отображается в память. Адрес этой структуры данных хранится в поле `MdlAddress` пакета IRP.
4. В этот момент драйвер получает вызов своей функции диспетчеризации. Пользовательский буфер, хотя он и заблокирован в памяти, недоступен из произвольного потока. Когда драйверу требуется доступ к буферу, он должен вызвать функцию, которая отображает тот же пользовательский буфер в системное адресное пространство, которое по умолчанию действительно в любом контексте процесса. Таким образом, фактически мы получаем два отображения для одного буфера: одно по исходному адресу (действительно только в контексте процесса, выдавшего запрос), другое в системном пространстве, которое действительно всегда. Для этого вызывается функция API `MmGetSystemAddressForMdlSafe`, которой передается структура MDL, построенная диспетчером ввода/вывода. Функция возвращает системный адрес.

5. После того как драйвер завершит запрос, диспетчер ввода/вывода удаляет второе отображение (в системное пространство), освобождает MDL и разблокирует пользовательский буфер, чтобы он мог выгружаться нормально, как любая другая память пользовательского режима.

На рис. 7.9, *а–е* изображена последовательность действий при прямом вводе/выводе.

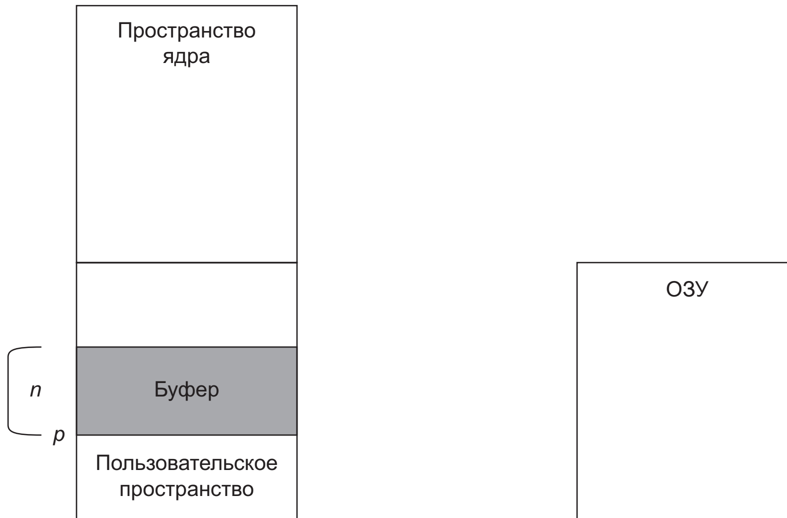


Рис. 7.9а. Прямой ввод/вывод: исходное состояние

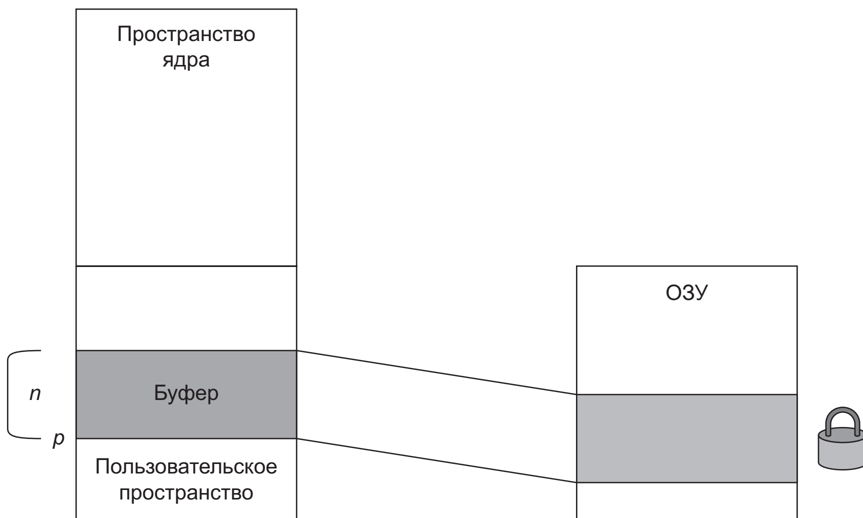


Рис. 7.9б. Прямой ввод/вывод: диспетчер ввода/вывода выводит страницы буфера в память и блокирует их

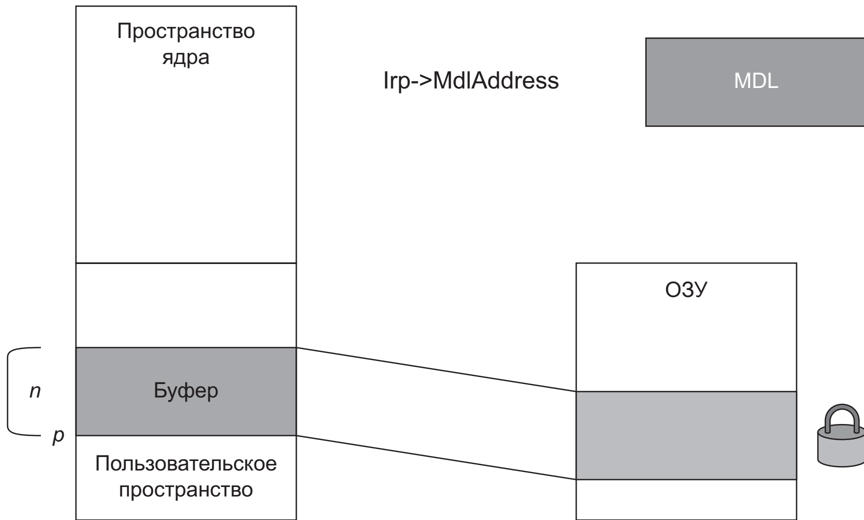


Рис. 7.9в. Прямой ввод/вывод: структура MDL, описывающая буфер, хранится в IRP

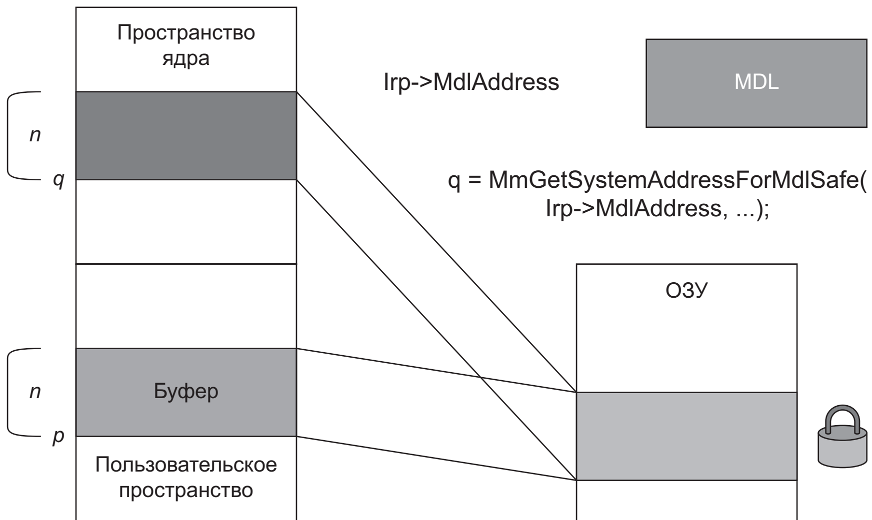


Рис. 7.9г. Прямой ввод/вывод: драйвер создает повторное отображение буфера в системное адресное пространство

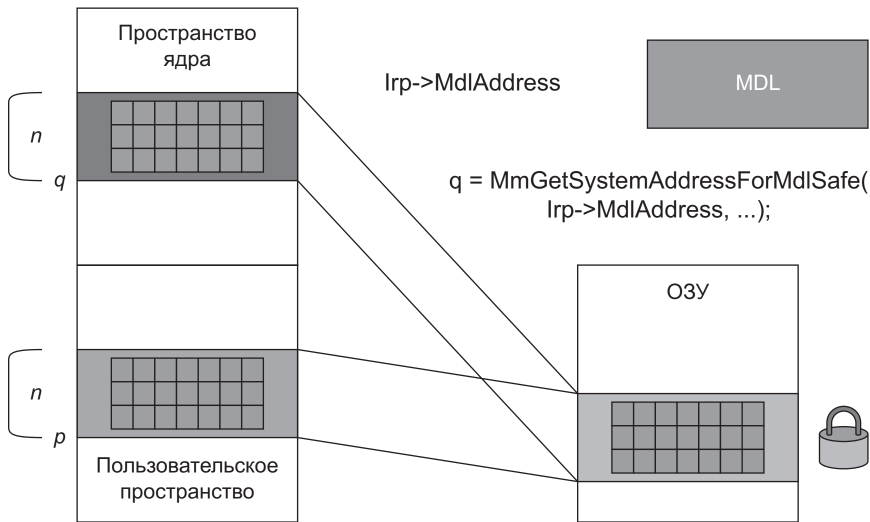


Рис. 7.9д. Прямой ввод/вывод: драйвер обращается к буферу по системному адресу

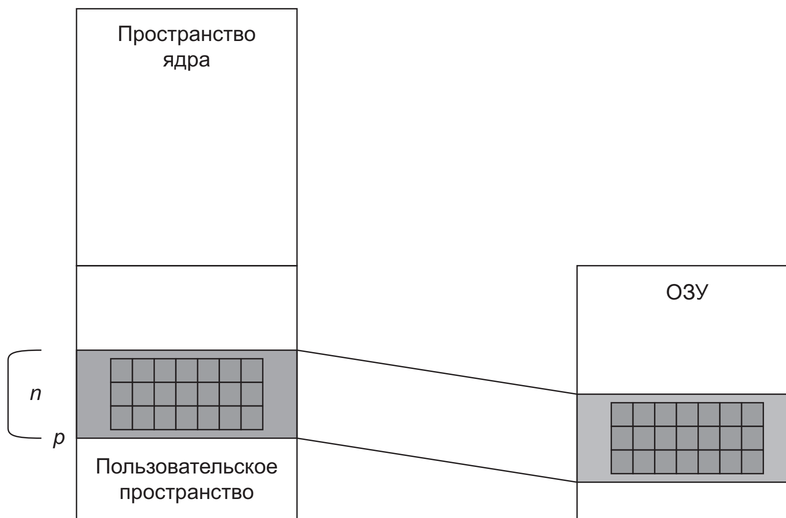


Рис. 7.9е. Прямой ввод/вывод: при завершении IRP диспетчер ввода/вывода удаляет отображение, удаляет MDL и разблокирует буфер

Обратите внимание: никакого копирования не происходит. Драйвер просто читает/записывает данные в пользовательский буфер напрямую, используя адрес системного пространства.

Пользовательский буфер блокируется функцией API MmProbeAndLockPages, полностью документированной в WDK. Разблокировка выполняется функцией MmUnlockPages (также документированной). Это означает, что драйвер может использовать эти функции за пределами узкого контекста прямого ввода/вывода.

Функция MmGetSystemAddressForMdlSafe может вызываться многократно. MDL хранит флаг, показывающий, было ли уже выполнено отображение из системного пространства. В таком случае достаточно вернуть существующий указатель.

MmGetSystemAddressForMdlSafe получает MDL и приоритет страницы (перечисление MM_PAGE_PRIORITY). Большинство драйверов использует NormalPagePriority, но также существуют значения LowPagePriority и HighPagePriority. Этот приоритет дает системе некоторое представление о важности отображения. За дополнительной информацией обращайтесь к документации WDK.

Если вызов MmGetSystemAddressForMdlSafe завершится неудачей, функция возвращает NULL. Это означает, что таблицы системных страниц отсутствуют или находятся в крайнем дефиците (в зависимости от заданного аргумента приоритета). Такая ситуация должна встречаться редко, но она возможна в ситуациях с крайней нехваткой памяти Драйвер должен проверить эту возможность; если возвращается NULL, то драйвер должен завершить IRP со статусом STATUS_INSUFFICIENT_RESOURCES.

Существует похожая функция MmGetSystemAddressForMdl, которая в случае неудачи инициирует фатальный сбой системы. Не пользуйтесь этой функцией.

Драйверы, которые не устанавливают ни один из флагов DO_BUFFERED_IO или DO_DIRECT_IO в объекте устройства, неявно используют обычный ввод/вывод (это означает, что драйвер не получает никакой специальной помощи от диспетчера ввода/вывода, а сам должен обеспечить работу с пользовательским буфером.

Пользовательские буферы для запросов IRP_MJ_DEVICE_CONTROL

В последних двух разделах рассматривались механизмы буферизованного и прямого ввода/вывода, а также их роль в запросах на чтение и запись. Для запросов IRP_MJ_DEVICE_CONTROL метод буферизации предоставляется на уровне кодов управляющих операций. Напомню, как выглядит прототип функции

`DeviceIoControl` пользовательского режима (эта функция похожа на функцию `ZwDeviceIoControlFile` режима ядра):

```
BOOL DeviceIoControl(
    HANDLE hDevice,              // Дескриптор устройства или файла
    DWORD dwIoControlCode,       // Код IOCTL (см. <winioctl.h>)
    PVOID lpInBuffer,            // Входной буфер
    DWORD nInBufferSize,         // Размер входного буфера
    PVOID lpOutBuffer,           // Выходной буфер
    DWORD nOutBufferSize,        // Размер выходного буфера
    PDWORD lpdwBytesReturned,    // Количество фактически возвращенных байтов
    LPOVERLAPPED lpOverlapped); // Для асинхронной операции
```

Центральное место занимают три аргумента: код управляющей операции ввода/вывода и два необязательных буфера: «входной» и «выходной». Оказывается, способ обращения к этим буферам зависит от кода управляющей операции, что очень удобно, потому что у разных запросов могут действовать разные требования относительно обращения к пользовательским буферам.

В главе 4 уже было показано, что код управляющей операции состоит из четырех аргументов, передаваемых макросу `CTL_CODE` — повторю его для удобства:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

Третий аргумент (`Method`) играет ключевую роль при выборе метода буферизации для обращения к входному и выходному буферам, предоставляемым с `DeviceIoControl`. Доступные варианты:

- `METHOD_NEITHER` — помощь со стороны диспетчера ввода/вывода не нужна, поэтому драйверу приходится организовывать работу с буфером самостоятельно. Например, этот режим может быть полезен, если для конкретного кода буфер не нужен (сам код управляющей операции содержит всю необходимую информацию), и лучше сообщить диспетчеру ввода/вывода, что никакая дополнительная работа не нужна.
    - В этом случае указатель на пользовательский входной буфер хранится в поле `Parameters.DeviceIoControl.Type3InputBuffer` текущей позиции стека, а выходной буфер — в поле IRP `UserBuffer`.
- `METHOD_BUFFERED` — для входного и выходного буфера используется буферизованный ввод/вывод. В начале запроса диспетчер ввода/вывода выделяет в невыгружаемом пуле системный буфер, размер которого равен максимуму из размеров входного и выходного буфера. Затем входной буфер копируется в системный буфер, и только после этого активизируется функция диспетчеризации `IRP_MJ_DEVICE_CONTROL`. После завершения запроса диспетчер ввода/вывода копирует количество байтов, заданное полем `IoStatus.Information` из IRP, в выходной пользовательский буфер.

- Указатель на системный буфер хранится в обычном месте: поле `AssociatedIrp.SystemBuffer` структуры IRP.

✦ `METHOD_IN_DIRECT` и `METHOD_OUT_DIRECT` — вопреки интуитивным ожиданиям, оба значения в отношении буферизации означают одно и то же: входной буфер использует буферизованный ввод/вывод, а выходной буфер использует прямой ввод/вывод. Единственное различие между двумя значениями заключается в том, доступен ли выходной буфер для чтения (`METHOD_IN_DIRECT`) или для записи (`METHOD_OUT_DIRECT`).

Последний пункт означает, что выходной буфер также может интерпретироваться как входной с использованием `METHOD_IN_DIRECT`.

В табл. 7.1 приведена сводка методов буферизации.

Таблица 7.1. Методы буферизации, соответствующие аргументу Method кода управляющей операции

Метод	Входной буфер	Выходной буфер
METHOD_NEITHER	–	–
METHOD_BUFFERED	Буферизованный	Буферизованный
METHOD_IN_DIRECT	Буферизованный	Прямой
METHOD_OUT_DIRECT	Буферизованный	Прямой

Всё вместе: драйвер Zero

В этом разделе вся информация, приведенная в этой главе (и предыдущих главах), будет использована для построения драйвера и клиентского приложения. Драйвер с именем Zero обладает следующими характеристиками:

✦ Для запросов на чтение он обнуляет переданный буфер.

✦ Для запросов на запись он просто потребляет переданный буфер по аналогии с классическим устройством *null*.

Драйвер будет использовать прямой ввод/вывод, чтобы избежать затрат на копирование, так как буферы, переданные клиентом, теоретически могут быть очень большими.

Работа над проектом начинается с создания пустого проекта WDM (Empty WDM Project) в Visual Studio и присваивания ему имени Zero. Удалите сгенерированный INF-файл.

Использование предварительно откомпилированного заголовка

Один из полезных приемов, который не относится исключительно к разработке драйверов, — использование предварительно откомпилированных заголовков. Эта возможность Visual Studio ускоряет компиляцию. Предварительно откомпилированный заголовок представляет собой заголовочный файл, который содержит команды `#include` для редко изменяемых заголовков (например, ntddk.h для драйверов). Предварительно откомпилированный заголовок компилируется однократно, сохраняется во внутреннем двоичном формате и используется при последующих компиляциях, которые выполняются заметно быстрее.

Многие проекты пользовательского режима, создаваемые в Visual Studio, уже используют предварительно откомпилированные заголовки. Так как мы начинаем с пустого проекта, предварительно откомпилированные заголовки придется настроить вручную. Выполните следующие действия, чтобы создать и использовать предварительно откомпилированный заголовок:

- Добавьте в проект новый файл заголовка с именем pch.h. Этот файл станет предварительно откомпилированным заголовком. Включите в него все редко изменяемые `#include`:

```
#pragma once

#include <ntddk.h>
```

- Добавьте исходный файл с именем pch.cpp и включите в него одну директиву `#include` — сам предварительно откомпилированный заголовок:

```
#include "pch.h"
```

- А теперь самая неочевидная часть — нужно сообщить компилятору, что pch.h является предварительно откомпилированным заголовком. Откройте окно свойств проекта, выберите пункт All Configurations and All Platforms, чтобы вам не пришлось настраивать все конфигурации/платформы по отдельности, перейдите в раздел C/C++ / Precompiled Headers, задайте параметру Precompiled Header значение Use, а параметру Precompiled Header File — значение pch.h (рис. 7.10). Щелкните на кнопке OK и закройте диалоговое окно.
- Файл pch.cpp должен быть назначен создателем предварительно откомпилированного заголовка. Щелкните правой кнопкой мыши на этом файле на панели Solution Explorer, выберите команду Properties. Перейдите в раздел C/C++/Precompiled Headers и присвойте параметру Precompiled Header значение Create (рис. 7.11). Щелкните на кнопке OK, чтобы подтвердить назначение.

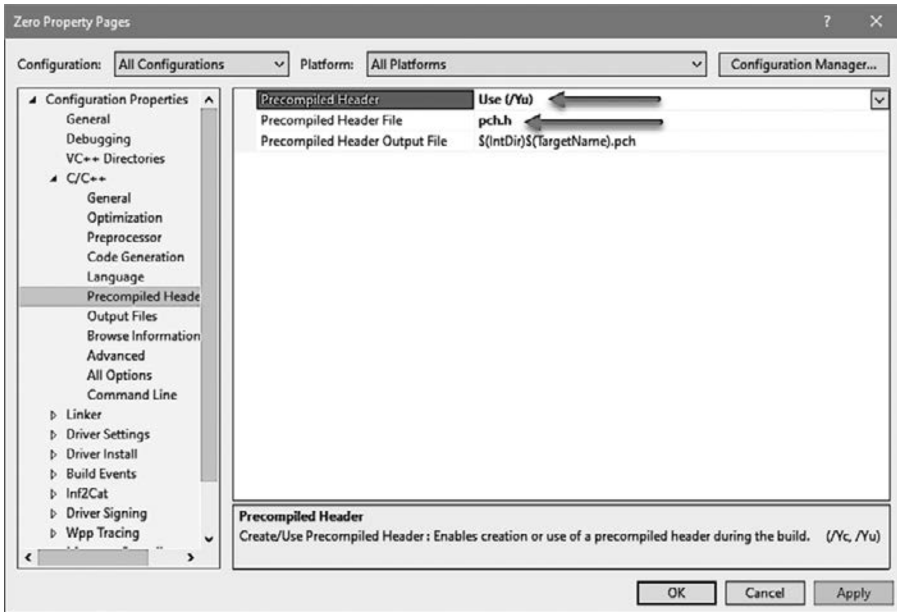


Рис. 7.10. Назначение предварительно откомпилированного заголовка для проекта

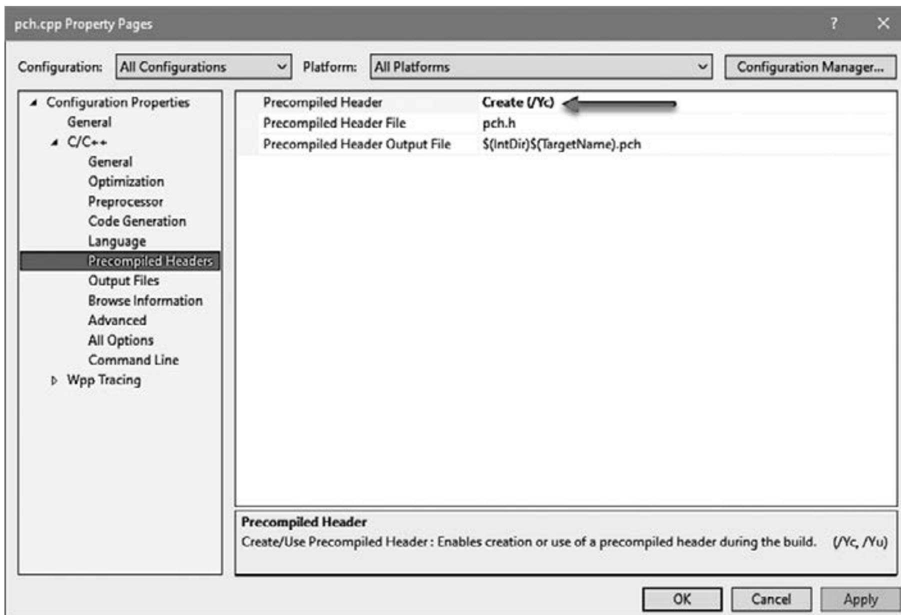


Рис. 7.11. Назначение предварительно откомпилированного заголовка для файла pch.cpp

С этого момента каждый файл C/CPP в проекте должен начинаться с директивы `#include "pch.h"`. Без этого проект компилироваться не будет.

Проследите за тем, чтобы перед директивой `#include "pch.h"` в исходном файле ничего не было. Все, что предшествует этой строке, вообще не компилируется!

Функция DriverEntry

Функция `DriveEntry` драйвера Zero очень похожа на ту, которая была написана для драйвера в главе 4. Тем не менее в драйвере главы 4 код отменяет операцию, которая была уже выполнена, в случае последующей ошибки. В драйвере было всего две операции, которые могли потребовать отмены: создание объекта устройства и создание символической ссылки. Драйвер Zero устроен похожим образом, но мы создадим более надежный код обработки ошибок в процессе инициализации. Начнем с азов: настройки функции выгрузки и функций диспетчеризации:

```
#define DRIVER_PREFIX "Zero: "

// DriverEntry

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = ZeroUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] = DriverObject->MajorFunction
                                                 [IRP_MJ_\
CLOSE] = ZeroCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = ZeroRead;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = ZeroWrite;
```

Теперь необходимо создать объект устройства и символическую ссылку, а также реализовать более общий и надежный механизм обработки ошибок. Для этого мы воспользуемся блоком `do/while(false)`, который в действительности не является циклом, но позволяет выйти из блока простой командой `break` в том случае, если что-то пошло не так:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\Zero");
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Zero");
PDEVICE_OBJECT DeviceObject = nullptr;
auto status = STATUS_SUCCESS;

do {
    status = IoCreateDevice(DriverObject, 0, &devName, FILE_DEVICE_UNKNOWN,
        0, FALSE, &DeviceObject);
```

```
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n",
            status));
        break;
    }
    // Включение прямого ввода/вывода
    DeviceObject->Flags |= DO_DIRECT_IO;

    status = IoCreateSymbolicLink(&symLink, &devName);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create symbolic link (0x%08X)\n",
        status));
        break;
    }
} while (false);

if (!NT_SUCCESS(status)) {
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    }
    return status;
```

Общая схема проста: если при любом вызове происходит ошибка, «цикл» просто прерывается командой `break`. За пределами цикла проверяется статус, и если произошла ошибка, отменяются все операции, выполненные до настоящего момента. С такой схемой легко добавить новые инициализации (которые понадобятся в более сложных драйверах), при этом код завершения хорошо локализуется и находится только в одном месте.

> Вместо решения с `do/while(false)` можно использовать команды `goto`, но как писал великий Дейкстра, «goto считается вредным», поэтому я стараюсь по возможности избегать этой конструкции.

Кроме того, устройство инициализируется для использования прямого ввода/вывода при операциях чтения и записи.

Функция диспетчеризации для чтения

Прежде чем переходить к реальной функции диспетчеризации для чтения, напишем вспомогательную функцию, которая упрощает завершение IRP с заданным статусом и информацией:

```
NTSTATUS CompleteIrp(PIRP Irp, NTSTATUS status = STATUS_SUCCESS,
                               ULONG_PTR info = 0)\
{
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, 0);
```

```
    return status;
}
```

Перейдем к реализации настоящей функции диспетчеризации. Сначала необходимо проверить длину буфера и убедиться в том, что она отлична от нуля. Если же она равна нулю, IRP просто завершается со статусом ошибки:

```
NTSTATUS ZeroRead(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Read.Length;
    if (len == 0)
        return CompleteIrp(Irp, STATUS_INVALID_BUFFER_SIZE);
```

Обратите внимание: длина пользовательского буфера передается через структуру `Parameters.Read` в текущей позиции стека ввода/вывода.

Так как для операции был настроен прямой ввод/вывод, заблокированный буфер необходимо отобразить в системное пространство функцией `MmGetSystemAddressForMdlSafe`:

```
auto buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress, NormalPagePriority);
if (!buffer)
    return CompleteIrp(Irp, STATUS_INSUFFICIENT_RESOURCES);
```

Функциональность, которую необходимо реализовать в драйвере, — заполнение нулями пользовательского буфера. Мы воспользуемся простым вызовом `memset` для заполнения буфера нулями, после чего завершим запрос:

```
    memset(buffer, 0, len);

    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

Важно присвоить полю `Information` длину буфера. Тем самым клиенту сообщается количество байтов, потребленных в ходе операции (в предпоследнем аргументе `ReadFile`). И это все, что необходимо сделать для операции чтения.

Функция диспетчеризации для записи

Функция диспетчеризации для записи еще проще. Все, что в ней необходимо сделать, — просто завершить запрос с длиной буфера, предоставленной клиентом (то есть, по сути, поглотить буфер):

```
NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;

    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

В данном случае мы даже не вызываем `MmGetSystemAddressForMdlSafe`, так как обращаться к фактическому буферу не нужно. По этой же причине этот вызов не делается заранее диспетчером ввода/вывода: возможно, она вообще не понадобится драйверу, а может быть, понадобится в определенных ситуациях; по этой причине диспетчер ввода/вывода подготавливает все (MDL) и дает возможность драйверу решить, когда выполнять отображение и выполнять ли его вообще.

Тестовое приложение

Добавим в решение проект нового консольного приложения для тестирования операций чтения и записи.

Простой код для тестирования этих операций:

```
int Error(const char* msg) {
    printf("%s: error=%d\n", msg, ::GetLastError());
    return 1;
}

int main() {
    HANDLE hDevice = ::CreateFile(L"\\\\.\\Zero", GENERIC_READ | GENERIC_WRITE,
        0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE) {
        return Error("failed to open device");
    }

    // Тестирование чтения
    BYTE buffer[64];

    // Сохранение ненулевых данных
    for (int i = 0; i < sizeof(buffer); ++i)
        buffer[i] = i + 1;

    DWORD bytes;
    BOOL ok = ::ReadFile(hDevice, buffer, sizeof(buffer), &bytes, nullptr);
    if (!ok)
        return Error("failed to read");
    if (bytes != sizeof(buffer))
        printf("Wrong number of bytes\n");

    // Проверить, равна ли сумма данных в буфере нулю
    long total = 0;
    for (auto n : buffer)
        total += n;
    if (total != 0)
        printf("Wrong data\n");

    // Тестирование записи
    BYTE buffer2[1024];      // Мусор
```

```
    ok = ::WriteFile(hDevice, buffer2, sizeof(buffer2), &bytes, nullptr);
    if (!ok)
        return Error("failed to write");
    if (bytes != sizeof(buffer2))
        printf("Wrong byte count\n");
    ::CloseHandle(hDevice);
}
```

Добавьте в драйвер следующую функциональность: драйвер должен подсчитывать общее количество байтов, передаваемых операциям чтения и записи. Драйвер предоставляет код управляющей операции, которая позволяет клиентскому коду запросить общее количество байтов, прочитанных и записанных с момента загрузки драйвера.

Решение этого упражнения (а также полный код всех проектов) доступно на странице книги на сайте GitHub по адресу https://github.com/zodiacon/windowskernelprogrammingbook.

Итоги

В этой главе мы научились обрабатывать пакеты IRP, с которыми постоянно приходится иметь дело драйверам. Вооружившись этой информацией, мы сможем пользоваться большей частью функциональности ядра. В следующей главе речь пойдет об обратных вызовах потоков и процессов.

Глава 8

Уведомления потоков и процессов

Один из мощных механизмов, доступных для драйверов режима ядра — возможность уведомления о некоторых важных событиях. В этой главе будут рассмотрены некоторые из этих событий, а именно создание и уничтожение процессов, создание и уничтожение потоков и загрузка образов.

В этой главе:

- Уведомления процессов
- Реализация уведомления процессов
- Передача данных в пользовательский режим
- Уведомления потоков
- Уведомления о загрузке образов
- Упражнения

Уведомления процессов

Каждый раз, когда в системе создается или уничтожается процесс, ядро может уведомить об этом факте заинтересованные драйверы. Это позволяет драйверам отслеживать состояние процессов (возможно, связывая с процессами некоторые данные). Как минимум это позволяет драйверам отслеживать создание/уничтожение процессов в реальном времени. Под «реальным временем» я имею в виду, что уведомления отправляются в оперативном режиме как часть создания процесса; драйвер не пропустит никакие процессы при создании и уничтожении.

При создании процесса драйвер также получает возможность остановить создание процесса и вернуть ошибку стороне, инициировавшей создание процесса. Эта возможность доступна только в режиме ядра.

> Windows предоставляет другие механизмы уведомления о создании или уничтожении процессов. Например, с механизмом ETW (Event Tracing for Windows) такие уведомления могут приниматься процессами пользовательского режима (работающими с повышенными привилегиями). Впрочем, предотвратить создание процесса при этом не удастся. Более того, у ETW существует внутренняя задержка уведомлений около 1–3 секунд (по причинам, связанным с быстродействием), так что процесс с коротким жизненным циклом может завершиться до получения уведомления. Если в этот момент будет сделана попытка открыть дескриптор для созданного процесса, произойдет ошибка.

Основная функция API для регистрации уведомлений процессов PsCreateSetProcessNotifyRoutineEx определяется так:

```
NTSTATUS
PsSetCreateProcessNotifyRoutineEx (
    _In_ PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,
    _In_ BOOLEAN Remove);
```

В настоящее время существует общесистемное ограничение на 64 регистрации, поэтому теоретически попытка регистрации может завершиться неудачей.

В первом аргументе передается функция обратного вызова драйвера, прототип которой выглядит так:

```
typedef void
(*PCREATE_PROCESS_NOTIFY_ROUTINE_EX) (
    _Inout_ PEPROCESS Process,
    _In_ HANDLE ProcessId,
    _Inout_opt_ PPS_CREATE_NOTIFY_INFO CreateInfo);
```

Второй аргумент PsCreateSetProcessNotifyRoutineEx указывает, что делает драйвер — регистрирует обратный вызов или отменяет его регистрацию (FALSE — первое). Обычно драйвер вызывает эту функцию с аргументом FALSE в своей функции DriverEntry, а потом вызывает ту же функцию с аргументом TRUE в своей функции выгрузки.

Аргументы функции уведомления:

- Process — объект создаваемого или уничтожаемого процесса.
- ProcessId — уникальный идентификатор процесса. Хотя аргумент объявлен с типом HANDLE, на самом деле это идентификатор.
- CreateInfo — структура с подробной информацией о создаваемом процессе. Если процесс уничтожается, то этот аргумент равен NULL.

При создании процесса функция обратного вызова драйвера выполняется создающим потоком. При выходе из процесса функция обратного вызова вы-

полняется последним потоком, выходящим из процесса. В обоих случаях обратный вызов вызывается в критической секции (с блокировкой нормальных APC-вызовов режима ядра).

В Windows 10 версии 1607 появилась другая функция для уведомлений процессов: `PsCreateSetProcessNotifyRoutineEx2`. Эта «расширенная» функция создает обратный вызов, сходный с предыдущим, но обратный вызов также активизируется для процессов Pico. Процессы Pico используются хост-процессами Linux для WSL (Windows Subsystem for Linux). Если драйвер заинтересован в таких процессах, он должен регистрироваться с расширенной функцией.

У драйвера, использующего эти обратные вызовы, должен быть установлен флаг `IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY` в заголовке PE (Portable Executable). Без установки флага вызов функции регистрации возвращает `STATUS_ACCESS_DENIED` (значение не имеет отношения к режиму тестовой подписи драйверов). В настоящее время Visual Studio не предоставляет пользовательского интерфейса для установки этого флага. Он должен задаваться в параметрах командной строки компоновщика ключом `/integritycheck`. На рис. 8.1 показаны свойства проекта при указании этого ключа.

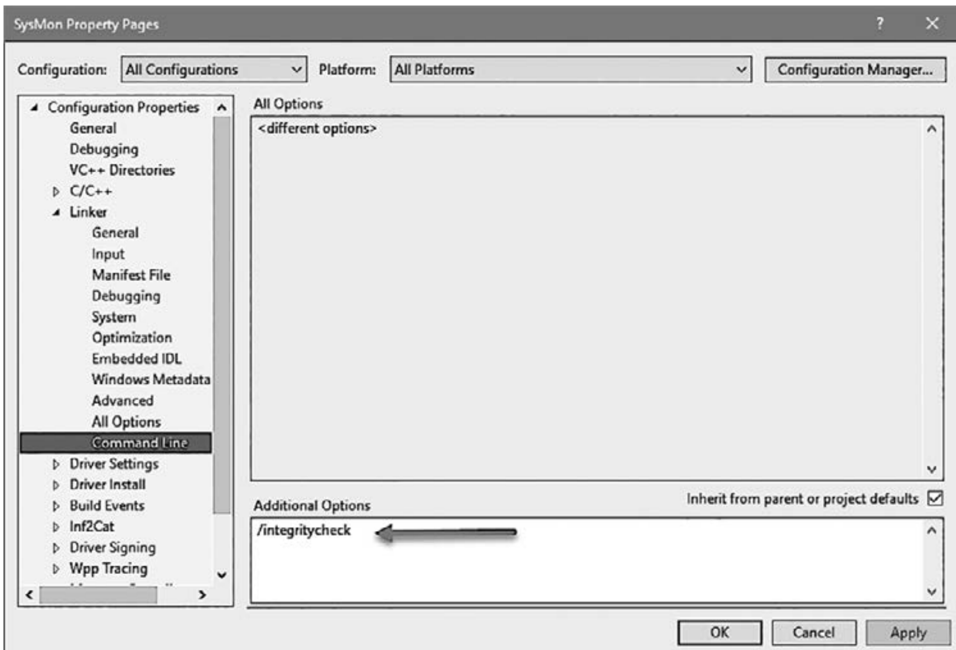


Рис. 8.1. Флаг компоновщика /integritycheck

Структура данных, предоставляемая для создания процесса, определяется следующим образом:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    _In_ SIZE_T Size;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG FileOpenNameAvailable : 1;
            _In_ ULONG IsSubsystemProcess : 1;
            _In_ ULONG Reserved : 30;
        };
    };
    _In_ HANDLE ParentProcessId;
    _In_ CLIENT_ID CreatingThreadId;
    _Inout_ struct _FILE_OBJECT *FileObject;
    _In_ PCUNICODE_STRING ImageFileName;
    _In_opt_ PCUNICODE_STRING CommandLine;
    _Inout_ NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

Описание важнейших полей этой структуры:

- `CreatingThreadId` — комбинация идентификаторов потока и процесса, вызывающего функцию создания процесса.
- `ParentProcessId` — идентификатор родительского процесса (не дескриптор). Этот процесс может быть тем же, который предоставляется `CreateThreadId.UniqueProcess`, но может быть и другим, так как при создании процесса может быть передан другой родитель, от которого будут наследоваться некоторые свойства.
- `ImageFileName` — имя файла с исполняемым образом; доступен при установленном флаге `FileOpenNameAvailable`.
- `CommandLine` — полная командная строка, используемая для создания процесса. Учтите, что он может быть равен `NULL`.
- `IsSubsystemProcess` — этот флаг устанавливается, если процесс является процессом Pico. Это возможно только в том случае, если драйвер регистрируется `PsCreateSetProcessNotifyRoutineEx2`.
- `CreationStatus` — статус, который будет возвращен вызывающей стороне. Драйвер может остановить создание процесса, поместив в это поле статус ошибки (например, `STATUS_ACCESS_DENIED`).

В обратных вызовах уведомления процессов следует применять защитное программирование. В частности, перед фактическим обращением необходимо проверять, что каждый указатель, по которому вы собираетесь обратиться, отличен от NULL.

Реализация уведомлений процессов

Чтобы продемонстрировать, как работают уведомления процессов, мы построим драйвер, который будет собирать информацию о создании и уничтожении процессов и предоставлять эту информацию для потребления клиентом пользовательского режима. Как и программа Process Monitor из пакета Sysinternals, он использует уведомления процессов (и потоков) для передачи информации об активности процессов (и потоков). В процессе реализации этого драйвера будут использованы некоторые средства, описанные в предыдущих главах.

Наш драйвер будет называться SysMon (хотя он никак не связан с программой SysMon из пакета Sysinternals). Он будет хранить всю информацию о создании/уничтожении в связном списке (с использованием структур `LIST_ENTRY`). Так как к связному списку могут одновременно обращаться несколько потоков, необходимо защитить его мьютексом или быстрым мьютексом; мы воспользуемся быстрым мьютексом, так как он более эффективен.

Собранные данные должны быть переданы в пользовательский режим, поэтому мы должны объявить стандартные структуры, которые будут строиться драйвером и получаться клиентом пользовательского режима. Мы добавим в проект драйвера стандартный заголовочный файл с именем SysMonCommon.h и определим несколько структур. Начнем со стандартного заголовка для всех информационных структур, который определяется следующим образом:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit
};

struct ItemHeader {
    ItemType Type;
    USHORT Size;
    LARGE_INTEGER Time;
};
```

Приведенное выше определение перечисления `ItemType` использует новую возможность C++ 11 — перечисления с областью видимости (scoped enums). В таких перечислениях значения имеют область видимости (`ItemType` в данном случае). Также размер этих перечислений может быть отличен от `int` — `short` в данном случае. Если вы работаете на C, используйте классические перечисления или даже `#define`.

Структура `ItemHeader` содержит информацию, общую для всех типов событий: тип события, время события (выраженное в виде 64-разрядного целого числа) и размер полезных данных. Размер важен, так как каждое событие имеет собственную информацию. Если позднее вы захотите упаковать массив таких

событий и (допустим) предоставить его клиенту пользовательского режима, клиент должен знать, где заканчивается каждое событие и начинается новое.

При наличии такого общего заголовка можно создать другие структуры данных для конкретных событий. Начнем с простейшего — выхода из процесса:

```
struct ProcessExitInfo : ItemHeader {
    ULONG ProcessId;
};
```

Для события выхода из процесса существует только один интересный фрагмент информации (кроме заголовка) — идентификатор завершаемого процесса.

Если вы работаете на C, наследование вам недоступно. Впрочем, его можно имитировать — создайте первое поле типа `ItemHeader`, а затем добавьте конкретные поля; структура памяти остается одинаковой.

```
struct ExitProcessInfo {
    ItemHeader Header;
    ULONG ProcessId;
};
```

Для идентификатора процесса используется тип `ULONG`. Использовать тип `HANDLE` не рекомендуется, так как в пользовательском режиме он может создать проблемы. Кроме того, тип `DWORD` не используется, хотя в заголовках пользовательского режима тип `DWORD` (32-разрядное целое без знака) встречается часто. В заголовках `WDK` тип `DWORD` не определен. И хотя определить его явно нетрудно, лучше использовать тип `ULONG` — он означает то же самое, но определяется в заголовках как пользовательского режима, так и режима ядра.

Так как каждая структура должна храниться как часть связанного списка, каждая структура данных должна содержать экземпляр `LIST_ENTRY` со ссылками на следующий и предыдущий элементы. Так как объекты `LIST_ENTRY` не должны быть доступны из пользовательского режима, мы определим расширенные структуры, содержащие эти элементы, в отдельном файле, который не будет использоваться в пользовательском режиме.

В новом файле с именем SysMon.h определяется параметризованная структура, в которой хранится поле `LIST_ENTRY` с основной структурой данных:

```
template<typename T>
struct FullItem {
    LIST_ENTRY Entry;
    T Data;
};
```

Параметризованный класс используется для того, чтобы вам не приходилось создавать множество типов, по одному для каждого конкретного типа события.

Например, для события выхода из процесса может быть создана следующая структура:

```
struct FullProcessExitInfo {
    LIST_ENTRY Entry;
    ProcessExitInfo Data;
};
```

> Также возможно наследовать от `LIST_ENTRY`, а затем добавить структуру `ProcessExitInfo`. Но такое решение менее элегантно, так как наши данные не имеют никакого отношения к `LIST_ENTRY`, поэтому расширение — искусственный прием, которого следует избегать.

Тип `FullItem<T>` избавляет от хлопот с созданием этих отдельных типов.

Если вы используете С, то, естественно, решение с шаблонами будет недоступно, поэтому вам придется применить представленный структурный подход. Не буду снова упоминать С — всегда существует обходное решение, которым можно воспользоваться в случае необходимости.

Заголовок связанного списка должен где-то храниться. Мы создадим структуру данных для хранения всего глобального состояния драйвера (вместо набора отдельных переменных). Определение структуры выглядит так:

```
struct Globals {
    LIST_ENTRY ItemsHead;
    int ItemCount;
    FastMutex Mutex;
};
```

В определении используется тип `FastMutex`, который был разработан в главе 6. Также в определении встречается RAII-обертка `AutoLock` на С++ (тоже из главы 6).

Функция DriverEntry

Функция `DriverEntry` для драйвера SysMon похожа на одноименную функцию драйвера Zero из главы 7. В нее нужно добавить регистрацию уведомлений процессов и инициализацию объекта `Globals`:

```
Globals g_Globals;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    auto status = STATUS_SUCCESS;
```

```
    InitializeListHead(&g_Globals.ItemsHead);
    g_Globals.Mutex.Init();

    PDEVICE_OBJECT DeviceObject = nullptr;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
    bool symLinkCreated = false;

    do {
        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\sysmon");
        status = IoCreateDevice(DriverObject, 0, &devName,
            FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n",
            status));
            break;
        }
        DeviceObject->Flags |= DO_DIRECT_IO;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create sym link (0x%08X)\n",
                status));
            break;
        }
        symLinkCreated = true;

        // Регистрация для уведомлений процессов
        status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to register process callback\
(0x%08X)\n",
                status));
            break;
        }
    } while (false);

    if (!NT_SUCCESS(status)) {
        if (symLinkCreated)
            IoDeleteSymbolicLink(&symLink);
        if (DeviceObject)
            IoDeleteDevice(DeviceObject);
    }

    DriverObject->DriverUnload = SysMonUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;

    return status;
}
```

Функция диспетчеризации для чтения позднее будет использоваться для возвращения информации о событиях пользовательскому режиму.

Обработка уведомлений о выходе из процессов

В приведенном выше коде функция уведомления процессов называется `OnProcessNotify`, а ее прототип был представлен ранее в этой главе. Эта функция обратного вызова обрабатывает события создания и завершения процессов. Начнем с выхода из процессов, так как это событие намного проще создания процесса (как вы вскоре увидите). Общая схема функции обратного вызова выглядит так:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    if (CreateInfo) {
        // Создание процесса
    }
    else {
        // Завершение процесса
    }
}
```

В случае выхода из процесса есть только идентификатор процесса, который необходимо сохранить (наряду с данными заголовка, общими для всех событий). Сначала необходимо выделить память для всей структуры, представляющей событие:

```
auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(PagedPool,
    sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
    return;
}
```

Если попытка выделения памяти завершается неудачей, драйвер ничего сделать не сможет, поэтому он просто возвращает управление из функции обратного вызова.

Затем нужно заполнить общую информацию: время, тип и размер элемента. Получить все эти данные несложно:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessExit;
item.ProcessId = HandleToULong(ProcessId);
item.Size = sizeof(ProcessExitInfo);

PushItem(&info->Entry);
```

Сначала мы обращаемся к самому элементу данных (в обход `LIST_ENTRY`) через переменную `info`. Затем заполняется информация заголовка: тип элемента хорошо известен, так как текущей является ветвь, обрабатывающая уведомления о завершении процессов; время можно получить при помощи

функции `KeQuerySystemTimePrecise`, возвращающей текущее системное время (UTC, не местное время) в формате 64-разрядного целого числа, с отчетом от 1 января 1601 года. Наконец, размер элемента — величина постоянная, равная размеру структуры данных, предоставляемой пользователю (а не размеру `FullItem<ProcessExitInfo>`).

Функция API `KeQuerySystemTimePrecise` появилась в Windows 8. В более ранних версиях следует использовать функцию API `KeQuerySystemTime`.

Дополнительные данные при завершении процесса состоят из идентификатора процесса. В коде используется функция `HandleToULong` для корректного преобразования объекта `HANDLE` в 32-разрядное целое без знака.

А теперь остается добавить новый элемент в конец связного списка. Для этого мы определим функцию с именем `PushItem`:

```
void PushItem(LIST_ENTRY* entry) {
    AutoLock<FastMutex> lock(g_Globals.Mutex);
    if (g_Globals.ItemCount > 1024) {
        // Слишком много элементов, удалить самый старый
        auto head = RemoveHeadList(&g_Globals.ItemsHead);
        g_Globals.ItemCount--;
        auto item = CONTAINING_RECORD(head, FullItem<ItemHeader>, Entry);
        ExFreePool(item);
    }
    InsertTailList(&g_Globals.ItemsHead, entry);
    g_Globals.ItemCount++;
}
```

Сначала код захватывает быстрый мьютекс, так как функция может вызываться сразу несколькими потоками одновременно. Все дальнейшее делается под защитой быстрого мьютекса.

Кроме того, драйвер ограничивает количество элементов связного списка. Такая предосторожность необходима, потому что ничто не гарантирует, что клиент будет быстро потреблять эти события. Драйвер не должен допускать неограниченное потребление данных, так как это может повредить системе в целом. Значение 1024 выбрано совершенно произвольно. Правильнее было бы читать это число из раздела драйвера в реестре.

Реализуйте это ограничение с чтением из реестра в `DriverEntry`. Подсказка: используйте такие функции API, как `ZwOpenKey` или `IoOpenDeviceRegistryKey`, а также `ZwQueryValueKey`.

Если счетчик элементов превысил максимальное значение, самый старый элемент удаляется; фактически связанный список рассматривается как очередь (`RemoveHeadList`). При освобождении элемента его память должна быть

освобождена. Указателем на элемент не обязательно должен быть указатель, изначально использованный для выделения памяти (хотя в данном случае это так, потому что объект `LIST_ENTRY` стоит на первом месте в структуре `FullItem<>`), поэтому для получения начального адреса объекта `FullItem<>` используется макрос `CONTAINING_RECORD`. Теперь элемент можно освободить вызовом `ExFreePool`.

На рис. 8.2 изображена структура объектов `FullItem<T>`.

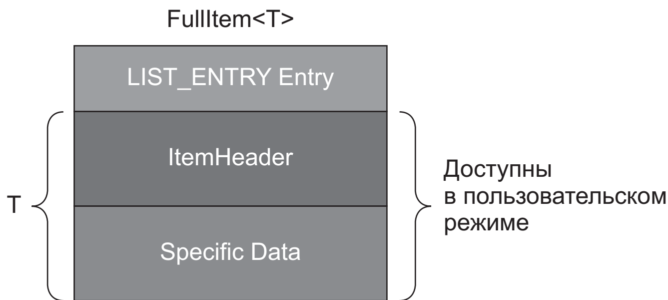


Рис. 8.2. Структура FullItem<T>

Наконец, драйвер вызывает `InsertTailList`, чтобы добавить элемент в конец списка, а счетчик элементов увеличивается на 1.

> Использовать атомарные операции инкремента/декремента в функции `PushItem` не обязательно, потому что операции со счетчиком элементов всегда выполняются под защитой быстрого мьютекса.

Обработка уведомлений о создании процессов

Обработка уведомлений о создании процессов создает больше проблем из-за непостоянного объема информации. Например, длина командной строки изменяется в зависимости от процесса. Сначала необходимо решить, какая информация должна сохраняться для создания процесса. Первая попытка:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    WCHAR CommandLine[1024];
};
```

В структуре сохраняется идентификатор процесса, идентификатор родительского процесса и командная строка. На первый взгляд такое решение работает и не создает проблем, потому что размер известен заранее.

Какие проблемы могут возникнуть при использовании приведенного определения?

Потенциальная проблема связана с командной строкой. Объявление командной строки с постоянным размером — решение простое, но проблематичное. Если командная строка окажется длиннее выделенного блока, драйвер будет вынужден произвести усечение (возможно, с потерей важной информации). Если командная строка короче выделенной, драйвер будет неэффективно расходовать память.

А можно ли использовать решение следующего вида:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    UNICODE_STRING CommandLine; // Будет работать?
};
```

Нет, такое решение работать не будет. Во-первых, `UNICODE_STRING` обычно не определяется в заголовках пользовательского режима. Во-вторых (что намного хуже), внутренний указатель на символы обычно будет указывать в системное пространство, недоступное для пользовательского режима.

Ниже приведен другой вариант, который мы используем в драйвере:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    USHORT CommandLineLength;
    USHORT CommandLineOffset;
};
```

В структуре будет храниться длина командной строки и ее смещение от начала структуры. Сами символы командной строки будут следовать за структурой в памяти. В этом случае мы не ограничиваем длину командной строки и не теряем память для коротких командных строк.

С таким объявлением можно приступить к построению реализации для создания процесса:

```
USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
USHORT commandLineSize = 0;
if (CreateInfo->CommandLine) {
    commandLineSize = CreateInfo->CommandLine->Length;
    allocSize += commandLineSize;
}
auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(PagedPool,
    allocSize, DRIVER_TAG);
```

```
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
return;
}
```

Суммарный размер выделяемого блока зависит от длины командной строки. Начнем с заполнения неизменяющейся информации, а именно заголовка, идентификаторов процесса и родительского процесса:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessCreate;
item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
item.ProcessId = HandleToULong(ProcessId);
item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
```

Размер элемента должен вычисляться с учетом базовой структуры и длины командной строки.

Затем необходимо скопировать командную строку по адресу за базовой структурой, а также обновить длину и смещение:

```
if (commandLineSize > 0) {
    ::memcpy((UCHAR*)&item + sizeof(item), CreateInfo->CommandLine->Buffer,
        commandLineSize);
    item.CommandLineLength = commandLineSize / sizeof(WCHAR); // Длина в WCHAR
    item.CommandLineOffset = sizeof(item);
}
else {
    item.CommandLineLength = 0;
}
PushItem(&info->Entry);
```

Добавьте в структуру `ProcessCreateInfo` имя файла образа по той же схеме, что и для командной строки. Будьте внимательны при вычислении смещения.

Передача данных в пользовательский режим

Затем следует понять, как передать собранную информацию клиенту пользовательского режима. Есть несколько возможных вариантов, но в нашем драйвере клиент будет запрашивать информацию у драйвера при помощи запроса чтения. Драйвер заполняет предоставленный буфер максимально возможным количеством событий (до исчерпания буфера или до последнего события в очереди).

Начнем обработку запроса чтения с получения адреса пользовательского буфера с применением прямого ввода/вывода (настраивается в `DriverEntry`):

```
NTSTATUS SysMonRead(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Read.Length;
    auto status = STATUS_SUCCESS;
    auto count = 0;
    NT_ASSERT(Irp->MdlAddress); // Используем прямой ввод/вывод

    auto buffer = (UCHAR*)MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
        NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
    else {
```

Теперь необходимо обратиться к связанному списку и извлечь элементы из заголовка:

```
AutoLock lock(g_Globals.Mutex); // C++ 17
while (true) {
    if (IsListEmpty(&g_Globals.ItemsHead)) // также можно проверить
                                           // g_Globals.ItemCount
        break;

    auto entry = RemoveHeadList(&g_Globals.ItemsHead);
    auto info = CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry);
    auto size = info->Data.Size;
    if (len < size) {
        // Пользовательский буфер заполнен, вставить элемент обратно
        InsertHeadList(&g_Globals.ItemsHead, entry);
        break;
    }

    g_Globals.ItemCount--;
    ::memcpy(buffer, &info->Data, size);
    len -= size;
    buffer += size;
    count += size;
    // Освободить данные после копирования
    ExFreePool(info);
}
```

Сначала мы захватываем быстрый мьютекс, так как уведомления процессов продолжают поступать. Если список пуст, то делать нечего, и выполнение цикла прерывается. После этого извлекается заголовочный элемент, и если его размер не превышает размер оставшейся части пользовательского буфера, копируется его содержимое (без поля `LIST_ENTRY`). Далее цикл продолжает извлекать элементы от заголовка списка, пока список не опустеет или пользовательский буфер не заполнится.

Наконец, запрос завершается с текущим статусом, а в поле `Information` сохраняется значение переменной `count`:

```
Irp->IoStatus.Status = status;
```

```
Irp->IoStatus.Information = count;
IoCompleteRequest(Irp, 0);
return status;
```

К функции выгрузки также стоит присмотреться повнимательнее. Если в связном списке присутствуют элементы, они должны быть освобождены явно; в противном случае возникнет утечка ресурсов:

```
void SysMonUnload(PDRIVER_OBJECT DriverObject) {
    // Отмена регистрации уведомлений процессов
    PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);

    // Освобождение оставшихся элементов
    while (!IsListEmpty(&g_Globals.ItemsHead)) {
        auto entry = RemoveHeadList(&g_Globals.ItemsHead);
        ExFreePool(CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry));
    }
}
```

Клиент пользовательского режима

После того как все будет готово, можно написать клиент пользовательского режима, который запрашивает данные вызовом `ReadFile` и выводит результаты.

Функция `main` вызывает `ReadFile` в цикле с небольшой приостановкой, чтобы поток не потреблял ресурсы процессора постоянно. Поступившие данные отправляются для вывода:

```
int main() {
    auto hFile = ::CreateFile(L"\\\\.\\SysMon", GENERIC_READ, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return Error("Failed to open file");

    BYTE buffer[1 << 16]; // 64-килобайтный буфер

    while (true) {
        DWORD bytes;
        if (!::ReadFile(hFile, buffer, sizeof(buffer), &bytes, nullptr))
            return Error("Failed to read");

        if (bytes != 0)
            DisplayInfo(buffer, bytes);

        ::Sleep(200);
    }
}
```

Функция `DisplayInfo` должна разобраться в структуре полученного буфера. Так как все события начинаются с общего заголовка, функция различает события по значению `ItemType`. После того как событие будет обработано, поле `Size` в заголовке указывает, где начинается следующее событие:

```
void DisplayInfo(BYTE* buffer, DWORD size) {
    auto count = size;
    while (count > 0) {
        auto header = (ItemHeader*)buffer;

        switch (header->Type) {
            case ItemType::ProcessExit:
            {
                DisplayTime(header->Time);
                auto info = (ProcessExitInfo*)buffer;
                printf("Process %d Exited\n", info->ProcessId);
                break;
            }

            case ItemType::ProcessCreate:
            {
                DisplayTime(header->Time);
                auto info = (ProcessCreateInfo*)buffer;
                std::wstring commandline((WCHAR*)(buffer +
                                                  info->CommandLineOffset),
                    info->CommandLineLength);
                printf("Process %d Created. Command line: %ws\n",
                                                          info->ProcessId,
                    commandline.c_str());
                break;
            }
            default:
                break;
        }
        buffer += header->Size;
        count -= header->Size;
    }
}
```

Для правильного извлечения командной строки в коде используется конструктор класса C++ `wstring`, который может построить строку по указателю и длине строки. Вспомогательная функция `DisplayTime` форматирует время в виде, удобном для чтения:

```
void DisplayTime(const LARGE_INTEGER& time) {
    SYSTEMTIME st;
    ::FileTimeToSystemTime((FILETIME*)&time, &st);
    printf("%02d:%02d:%02d.%03d: ",
        st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
}
```

Драйвер устанавливается и запускается так, как было описано в главе 4.

```
sc create sysmon type= kernel binPath= C:\Book\SysMon.sys

sc start sysmon
```

Пример вывода, полученного при запуске SysMonClient.exe:

```
C:\Book>SysMonClient.exe
12:06:24.747: Process 13000 Exited
12:06:31.032: Process 7484 Created. Command line: SysMonClient.exe
12:06:42.461: Process 3128 Exited
12:06:42.462: Process 7936 Exited
12:06:42.474: Process 12320 Created. Command line: "C:\$WINDOWS.~BT\Sources\
                                                       mighost.\
exe" {5152EFE5-97CA-4DE6-BBD2-4F6ECE2ABD7A} /InitDoneEvent:MigHost.
                                             {5152EFE5-97CA-4D\
E6-BBD2-4F6ECE2ABD7A}.Event /ParentPID:11908 /LogDir:"C:\$WINDOWS.~BT\Sources\
                                              Panthe\
r"
12:06:42.485: Process 12796 Created. Command line: \??\C:\WINDOWS\system32\
                                                   conhost.e\
xe 0xffffffff -ForceV1
12:07:09.575: Process 6784 Created. Command line: "C:\WINDOWS\system32\cmd.exe"
12:07:09.590: Process 7248 Created. Command line: \??\C:\WINDOWS\system32\
                                                   conhost.ex\
e 0xffffffff -ForceV1
12:07:11.387: Process 7832 Exited
12:07:12.034: Process 2112 Created. Command line: C:\WINDOWS\system32\
                                                  ApplicationFra\
meHost.exe -Embedding
12:07:12.041: Process 5276 Created. Command line: "C:\Windows\SystemApps\
                                                  Microsoft.M\
icrosoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe" -ServerName:MicrosoftEdge.
                                              AppXdnhjhccw\
3zf0j06tkg3jtqr00qdm0khc.mca
12:07:12.624: Process 2076 Created. Command line: C:\WINDOWS\system32\
                                                  DllHost.exe /P\
rocessid:{7966B4D8-4FDC-4126-A10B-39A3209AD251}
12:07:12.747: Process 7080 Created. Command line: C:\WINDOWS\system32\
                                                  browser_broker\
.exe -Embedding
12:07:13.016: Process 8972 Created. Command line: C:\WINDOWS\System32\
                                                  svchost.exe -k\
LocalServiceNetworkRestricted
12:07:13.435: Process 12964 Created. Command line: C:\WINDOWS\system32\
                                                   DllHost.exe /\
Processid:{973D20D7-562D-44B9-B70B-5A0F49CCDF3F}
12:07:13.554: Process 11072 Created. Command line: C:\WINDOWS\system32\
                                                   Windows.WARP.\
JITService.exe 7f992973-8a6d-421d-b042-6afd93a19631
               S-1-15-2-3624051433-2125758914-1\
423191267-1740899205-1073925389-3782572162-737981194
          S-1-5-21-4017881901-586210945-2\
```

```
666946644-1001 516
12:07:14.454: Process 12516 Created. Command line: C:\Windows\System32\
RuntimeBroker.exe -Embedding
12:07:14.914: Process 10424 Created. Command line: C:\WINDOWS\system32\
                                                    MicrosoftEdge\
SH.exe SCODEF:5276 CREDAT:9730 APH:1000000000000017 JITHOST /prefetch:2
12:07:14.980: Process 12536 Created. Command line: "C:\Windows\System32\
                                                    MicrosoftEdg\
eCP.exe" -ServerName:Windows.Internal.WebRuntime.ContentProcessServer
12:07:17.741: Process 7828 Created. Command line: C:\WINDOWS\system32\
                                                   SearchIndexer.\
exe /Embedding
12:07:19.171: Process 2076 Exited
12:07:30.286: Process 3036 Created. Command line: "C:\Windows\System32\
                                                   MicrosoftEdge\
CP.exe" -ServerName:Windows.Internal.WebRuntime.ContentProcessServer
12:07:31.657: Process 9536 Exited
```

Уведомления потоков

Ядро предоставляет обратные вызовы создания и уничтожения потоков, аналогичные обратным вызовам процессов. Для регистрации используется функция API `PsSetCreateThreadNotifyRoutine`, а для ее отмены — другая функция, `PsRemoveCreateThreadNotifyRoutine`. В аргументах функции обратного вызова передается идентификатор процесса, идентификатор потока, а также флаг создания/уничтожения потока.

Расширим существующий драйвер SysMon, чтобы он получал не только уведомления процессов, но и уведомления потоков. Начнем с добавления значений перечисления и структуры, представляющей информацию, — все это добавляется в заголовочный файл SysMonCommon.h:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit
};

struct ThreadCreateExitInfo : ItemHeader {
    ULONG ThreadId;
    ULONG ProcessId;
};
```

Затем можно добавить вызов регистрации в `DriverEntry`, непосредственно за вызовом регистрации уведомлений процессов:

```
status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set thread callbacks (status=%08X)\n",
status)\
);
    break;
}
```

Сама функция обратного вызова весьма проста, так как структура события имеет постоянный размер. Полный код функции обратного вызова для потока:

```
void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
    auto size = sizeof(FullItem<ThreadCreateExitInfo>);
    auto info = (FullItem<ThreadCreateExitInfo>*)ExAllocatePoolWithTag(PagedPool,
        size, DRIVER_TAG);
    if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
        return;
    }
    auto& item = info->Data;
    KeQuerySystemTimePrecise(&item.Time);
    item.Size = sizeof(item);
    item.Type = Create ? ItemType::ThreadCreate : ItemType::ThreadExit;
    item.ProcessId = HandleToULong(ProcessId);
    item.ThreadId = HandleToULong(ThreadId);

    PushItem(&info->Entry);
}
```

Большая часть кода выглядит довольно знакомо.

Чтобы завершить реализацию, мы добавим в клиент код для вывода информации о создании и уничтожении потоков (в `DisplayInfo`):

```
case ItemType::ThreadCreate:
{
    DisplayTime(header->Time);
    auto info = (ThreadCreateExitInfo*)buffer;
    printf("Thread %d Created in process %d\n",
        info->ThreadId, info->ProcessId);
    break;
}

case ItemType::ThreadExit:
{
    DisplayTime(header->Time);
    auto info = (ThreadCreateExitInfo*)buffer;
    printf("Thread %d Exited from process %d\n",
        info->ThreadId, info->ProcessId);
    break;
}
```

Пример вывода с обновленным драйвером и клиентом:

```
13:06:29.631: Thread 12180 Exited from process 11976
13:06:29.885: Thread 13016 Exited from process 8820
13:06:29.955: Thread 12532 Exited from process 8560
13:06:30.218: Process 12164 Created. Command line: SysMonClient.exe
13:06:30.219: Thread 12004 Created in process 12164
13:06:30.607: Thread 12876 Created in process 10728

...

13:06:33.260: Thread 4524 Exited from process 4484
13:06:33.260: Thread 13072 Exited from process 4484
13:06:33.263: Thread 12388 Exited from process 4484
13:06:33.264: Process 4484 Exited
13:06:33.264: Thread 4960 Exited from process 5776
13:06:33.264: Thread 12660 Exited from process 5776
13:06:33.265: Process 5776 Exited
13:06:33.272: Process 2584 Created. Command line: "C:\$WINDOWS.~BT\Sources\
                                                  mighost.e\
xe" {CCD9805D-B15B-4550-94FB-B2AE544639BF} /InitDoneEvent:MigHost.
                                            {CCD9805D-B15B-455\
0-94FB-B2AE544639BF}.Event /ParentPID:11908 /LogDir:"C:\$WINDOWS.~BT\Sources\
                                             Panther\
"
13:06:33.272: Thread 13272 Created in process 2584
13:06:33.280: Process 12120 Created. Command line: \??\C:\WINDOWS\system32\
                                                   conhost.e\
xe 0xffffffff -ForceV1
13:06:33.280: Thread 4200 Created in process 12120
13:06:33.283: Thread 4400 Created in process 12120
13:06:33.284: Thread 9632 Created in process 12120
13:06:33.284: Thread 6064 Created in process 12120
13:06:33.289: Thread 2472 Created in process 12120
```

Добавьте в клиент код вывода имени образа процесса при создании и завершении потока.

Уведомления о загрузке образов

Последний механизм обратного вызова, который будет рассмотрен в этой главе, — уведомления о загрузке образов. Каждый раз, когда в системе загружается файл образа (EXE, DLL, драйвер), драйвер может получать уведомление.

Функция API `PsSetLoadImageNotifyRoutine` регистрируется для получения этих уведомлений, а функция `PsRemoveImageNotifyRoutine` отменяет регистрацию. Функция обратного вызова имеет следующий прототип:

```
typedef void (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    _In_opt_ PUNICODE_STRING FullImageName,
    _In_ HANDLE ProcessId, // pid, с которым связывается образ
    _In_ PIMAGE_INFO ImageInfo);
```

> Любопытно, что парного механизма обратного вызова для уведомления о выгрузке образов не существует.

Аргумент `FullImageName` не так прост. Как указывает аннотация SAL, он необязателен и может содержать `NULL`. Но даже если он отличен от `NULL`, он не всегда содержит точное имя файла образа.

Причины кроются глубоко в ядре и выходят за рамки книги. В большинстве случаев решение работает нормально, а путь использует внутренний формат NT, начинающийся с «`\Device\HadrdiskVolumex\`…» вместо «`c:\`…». Преобразование может быть выполнено разными способами. Тема более подробно рассматривается в главе 11.

Аргумент `ProcessId` содержит идентификатор процесса, в котором загружается образ. Для драйверов (образов режима ядра) это значение равно нулю.

Аргумент `ImageInfo` содержит дополнительную информацию об образе; его объявление выглядит так:

```
#define IMAGE_ADDRESSING_MODE_32BIT 3

typedef struct _IMAGE_INFO {
    union {
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode  : 8; // Режим адресации
            ULONG SystemModeImage      : 1; // Образ системного режима
            ULONG ImageMappedToAllPids : 1; // Образ отображается во все процессы
            ULONG ExtendedInfoPresent  : 1; // Доступна структура IMAGE_INFO_EX
            ULONG MachineTypeMismatch  : 1; // Несоответствие типа архитектуры
            ULONG ImageSignatureLevel  : 4; // Уровень цифровой подписи
            ULONG ImageSignatureType   : 3; // Тип цифровой подписи
            ULONG ImagePartialMap      : 1; // Не равно 0 при частичном
                                                отображении
            ULONG Reserved : 12;
        };
    };
    PVOID    ImageBase;
    ULONG    ImageSelector;
    SIZE_T   ImageSize;
    ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Краткая сводка важных полей структуры:

- ✦ `SystemModeImage` — флаг устанавливается для образа режима ядра и сбрасывается для образа пользовательского режима.
- ✦ `ImageSignatureLevel` — уровень цифровой подписи (Windows 8.1 и выше). См. описание констант `SE_SIGNING_LEVEL_` в WDK.
- ✦ `ImageSignatureType` — тип сигнатуры (Windows 8.1 и выше). См. описание перечисления `SE_IMAGE_SIGNATURE_TYPE` в WDK.
- ✦ `ImageBase` — виртуальный адрес, по которому загружается образ.
- ✦ `ImageSize` — размер образа.
- ✦ `ExtendedInfoPresent` — если флаг установлен, `IMAGE_INFO` является частью большей структуры `IMAGE_INFO_EX`:

```
typedef struct _IMAGE_INFO_EX {
    SIZE_T Size;
    IMAGE_INFO ImageInfo;
    struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

Для обращения к большей структуре драйвер использует макрос `CONTAINING_RECORD`:

```
if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    // Обращение к FileObject
}
```

В расширенной структуре добавляется всего одно осмысленное поле — объект файла, используемый для управления образом. Драйвер может добавить ссылку на объект (`ObReferenceObject`) и использовать его в других функциях по мере надобности.

Добавьте в драйвер SysMon уведомления о загрузке образов; драйвер должен собирать информацию только для образов пользовательского режима. Клиент должен выводить путь образа, идентификатор процесса и базовый адрес образа.

Упражнения

1. Напишите драйвер, который отслеживает создание процессов и позволяет клиентскому приложению настроить пути к исполняемым файлам, для которых выполнение должно быть запрещено.

2. Напишите драйвер (или расширьте драйвер SysMon), который будет обнаруживать удаленное создание потоков, — то есть создание потоков в процессе, отличном от текущего. Подсказка: первый поток в процессе всегда создается «удаленно». Уведомите клиента пользовательского режима об этом событии. Напишите тестовое приложение, которое использует функцию `CreateRemoteThread` для тестирования.

Итоги

В этой главе были рассмотрены некоторые механизмы обратного вызова, предоставляемые ядром: уведомления процессов, потоков и образов. В следующей главе мы продолжим изучение механизмов обратного вызова — в ней будут рассмотрены уведомления объектов и реестра.

Глава 9
Уведомления объектов и реестра

Ядро предоставляет другие возможности перехвата некоторых операций. Начнем с уведомлений объектов, позволяющих узнавать о получении дескрипторов некоторых типов объектов. Затем будут рассмотрены возможности перехвата операций с реестром.

В этой главе:

- Уведомления объектов
- Драйвер Process Protector
- Уведомления реестра
- Реализация уведомлений реестра
- Упражнения

Уведомления объектов

Ядро предоставляет механизм уведомления заинтересованных драйверов о попытках открытия или дублирования дескрипторов некоторых типов объектов. Официально поддерживаемые типы объектов — процессы и потоки, а для Windows 10 — также рабочий стол.

Для регистрации используется функция API `ObRegisterCallbacks`, прототип которой выглядит так:

```
NTSTATUS ObRegisterCallbacks (
    _In_ POB_CALLBACK_REGISTRATION CallbackRegistration,
    _Outptr_ PVOID *RegistrationHandle);
```

До регистрации структура `OB_CALLBACK_REGISTRATION` должна быть инициализирована с передачей необходимой информации о том, на что именно реги-

стрируется драйвер. RegistrationHandle содержит возвращаемое значение при успешной регистрации; это всего лишь непрозрачный указатель, используемый для отмены регистрации вызовом ObUnRegisterCallbacks.

ОБЪЕКТЫ РАБОЧЕГО СТОЛА

Рабочий стол (desktop) — объект ядра, содержащийся в Window Station — еще одном объекте ядра, который сам по себе является частью объекта Session. Рабочий стол содержит окна, меню и перехватчики. В данном случае под «перехватчиками» (hooks) имеются в виду перехватчики пользовательского режима, используемые с функцией API SetWindowsHookEx.

Обычно при входе пользователя в систему создаются два рабочих стола. Рабочий стол с именем «Winlogon» создается процессом Winlogon.exe. Этот рабочий стол вы видите при нажатии комбинации клавиш SAS (Secure Attention Sequence) — чаще всего Ctrl+Alt+Del. Второй рабочий стол с именем «default» — обычный рабочий стол с обычными окнами, хорошо знакомый нам всем. Переключение на другой рабочий стол осуществляется функцией API SwitchDesktop. За дополнительной информацией обращайтесь к сообщению в блоге[1].

[1] https://scorpiosoftware.net/2019/02/17/windows-10-desktops-vs-sysinternals-desktops/.

Драйверы, использующие ObRegisterCallbacks, должны компоноваться с ключом /integritycheck.

Определение OB_CALLBACK_REGISTRATION:

```
typedef struct _OB_CALLBACK_REGISTRATION {
    _In_ USHORT Version;
    _In_ USHORT OperationRegistrationCount;
    _In_ UNICODE_STRING Altitude;
    _In_ PVOID RegistrationContext;
    _In_ OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

Version — поле, которому обязательно должна быть присвоена константа OB_FLT_REGISTRATION_VERSION (в настоящее время 0x100). Количество регистрируемых операций задается полем OperationRegistrationCount. Значение определяет количество структур OB_OPERATION_REGISTRATION, на которые указывает OperationRegistration. Каждая из них предоставляет информацию о типе объектов, представляющем интерес (процесс, поток или рабочий стол).

Изучим аргумент `Altitude`. Он определяет число (в строковом виде), которое влияет на порядок активизации обратных вызовов для этого драйвера. Это необходимо, потому что другие драйверы могут иметь свои обратные вызовы, и ответ на вопрос о том, какой драйвер будет активизирован первым, определяется аргументом `Altitude` — чем выше значение, тем ранее в цепочке будет активизирован драйвер.

Какое значение следует присвоить `Altitude`? В большинстве случаев это неважно, и выбор зависит от драйвера. Предоставленное значение `Altitude` не должно конфликтовать со значениями, заданными ранее зарегистрированными драйверами. Значение не обязано быть целым числом. На самом деле это вещественное число с бесконечной точностью (именно поэтому оно задается в строковом виде). Для предотвращения конфликтов значение `Altitide` должно задаваться со случайными числами в дробной части — например, «12345.1762389». Вероятность конфликта в таком случае ничтожна. Драйвер даже может генерировать случайные цифры для предотвращения конфликтов. Если попытка регистрации завершится неудачей со статусом `STATUS_FLT_INSTANCE_ALTITUDE_COLLISION`, это означает конфликт значений `Altitude`; внимательный разработчик драйвера должен скорректировать значение и повторить попытку.

> Концепция `Altitude` также используется для фильтрации реестра (см. раздел «Уведомления реестра» этой главы) и мини-фильтров файловой системы (см. следующую главу).

Наконец, `RegistrationContext` — определяемое драйвером значение, которое передается в неизменном виде функции(-ям) обратного вызова.

В структуре(-ах) `OB_OPERATION_REGISTRATION` драйвер настраивает свои обратные вызовы и определяет, какие типы объектов и операции представляют интерес. Структура определяется следующим образом:

```
typedef struct _OB_OPERATION_REGISTRATION {
    _In_ POBJECT_TYPE *ObjectType;
    _In_ OB_OPERATION Operations;
    _In_ POB_PRE_OPERATION_CALLBACK PreOperation;
    _In_ POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

`ObjectType` — указатель на тип объекта для регистрации экземпляра (процесс, поток или рабочий стол). Эти указатели экспортируются в виде глобальных переменных ядра: `PsProcessType`, `PsThreadType` и `ExDesktopObjectType` соответственно.

Поле `Operations` содержит перечисление битовых флагов для выбора операции создания/открытия (`OB_OPERATION_HANDLE_CREATE`) и/или дублирования

(`OB_OPERATION_HANDLE_DUPLICATE`). `OB_OPERATION_HANDLE_CREATE` обозначает вызовы функций пользовательского режима, таких как `CreateProcess`, `OpenProcess`, `CreateThread`, `OpenThread`, `CreateDesktop`, `OpenDesktop` и аналогичных функций для этих типов объектов. `OB_OPERATION_HANDLE_DUPLICATE` относится к дублированию дескрипторов для этих объектов (функция API пользовательского режима `DuplicateHandle`).

Для совершения одного из этих вызовов (также из режима ядра, кстати говоря) могут быть зарегистрированы один или два обратных вызова: перед операцией (поле `PreOperation`) и после операции (`PostOperation`).

Обратный вызов перед операцией

Обратный вызов перед операцией вызывается перед завершением операций создания/открытия/дублирования, предоставляя драйверу возможность внести изменения в результат операции. Обратный вызов перед операцией получает структуру `OB_PRE_OPERATION_INFORMATION`, которая определяется следующим образом:

```
typedef struct _OB_PRE_OPERATION_INFORMATION {
    _In_ OB_OPERATION Operation;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG KernelHandle:1;
            _In_ ULONG Reserved:31;
        };
    };
    _In_ PVOID Object;
    _In_ POBJECT_TYPE ObjectType;
    _Out_ PVOID CallContext;
    _In_ POB_PRE_OPERATION_PARAMETERS Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;
```

Краткая сводка полей структуры:

- `Operation` — определяет тип операции (`OB_OPERATION_HANDLE_CREATE` или `OB_OPERATION_HANDLE_DUPLICATE`).
- `KernelHandle` (внутри `Flags`) — указывает, что это дескриптор ядра. Дескрипторы ядра могут создаваться и использоваться только кодом режима ядра. Например, это позволяет драйверу игнорировать запросы ядра.
- `Object` — указатель на реальный объект, для которого создается/открывается/дублируется дескриптор. Для процессов это адрес `EPROCESS`, для потоков — адрес `PETHREAD`.
- `ObjectType` — указатель на тип объекта: `*PsProcessType`, `*PsThreadType` или `*ExDesktopObjectType`.

- CallContext — значение, определяемое драйвером, которое передается обратному вызову после операции для данного экземпляра.
- Parameters — объединение с дополнительной информацией, зависящей от Operation. Определяется следующим образом:

```
typedef union _OB_PRE_OPERATION_PARAMETERS {
    _Inout_ OB_PRE_CREATE_HANDLE_INFORMATION CreateHandleInformation;
    _Inout_ OB_PRE_DUPLICATE_HANDLE_INFORMATION DuplicateHandleInformation;
} OB_PRE_OPERATION_PARAMETERS, *POB_PRE_OPERATION_PARAMETERS;
```

Драйвер должен проверить соответствующее поле в зависимости от операции. Для операций Create драйвер получает следующую информацию:

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
    _Inout_ ACCESS_MASK DesiredAccess;
    _In_ ACCESS_MASK OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;
```

OriginalDesiredAccess — маска доступа, заданная на стороне вызова. Следующий код пользовательского режима открывает дескриптор для существующего процесса:

```
HANDLE OpenHandleToProcess(DWORD pid) {
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
        FALSE, pid);
    if(!hProcess) {
        // Не удалось открыть дескриптор
    }
    return hProcess;
}
```

В этом примере клиент пытается получить дескриптор для процесса с заданной маской доступа, описывающей его «намерения» по отношению к процессу. Функция обратного вызова перед операцией получает это значение в поле OriginalDesiredAccess. Значение также копируется в DesiredAccess. Обычно ядро определяет (на основании контекста безопасности и дескриптора безопасности процесса), можно ли предоставить клиенту запрашиваемый доступ.

Драйвер может на основании собственной логики изменить DesiredAccess для примера, исключив некоторые права доступа, запрашиваемые клиентом:

```
OB_PREOP_CALLBACK_STATUS OnPreOpenProcess(PVOID /* RegistrationContext */,
    POB_PRE_OPERATION_INFORMATION Info) {

    if(/* логика */) {
        Info->Parameters->CreateHandleInformation.DesiredAccess &=
~PROCESS_VM_READ;
    }
    return OB_PREOP_SUCCESS;
}
```

Приведенный фрагмент кода исключает маску доступа `PROCESS_VM_READ` перед тем, как разрешить нормальное продолжение операции. Если в конечном итоге операция завершится удачно, клиент получит обратно действительный дескриптор, но только с маской доступа `PROCESS_QUERY_INFORMATION`.

Полный список масок доступа процессов, потоков и рабочих столов см. в документации MSDN.

Невозможно добавить новые биты маски доступа, которые не были запрошены клиентом.

Для операций дублирования драйверу передается следующая информация:

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
    _Inout_ ACCESS_MASK DesiredAccess;
    _In_ ACCESS_MASK OriginalDesiredAccess;
    _In_ PVOID SourceProcess;
    _In_ PVOID TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, *POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

Поле `DesiredAccess` можно изменять, как и прежде. В дополнительной информации передается исходный процесс (из которого дублируется дескриптор) и целевой процесс (процесс, в который дублируется новый дескриптор). Это позволяет драйверу запросить различные свойства процессов, прежде чем принять решение относительно того, как изменить маску доступа (и нужно ли изменять ее вообще).

Как получить больше информации о процессе по его адресу? Так как структура EPROCESS не документирована, а экспортированных и документированных функций для прямой работы с такими указателями совсем немного, задача получения подробной информации кажется проблематичной. Также можно воспользоваться функцией `ZwQueryInformationProcess` для получения нужной информации, но этой функции необходим дескриптор, который можно получить вызовом `ObOpenObjectByPointer`. Этот прием более подробно рассматривается в главе 11.

Обратный вызов после операции

Обратные вызовы после операции активируются после завершения операции. В этот момент драйвер уже не может вносить никакие изменения, он может только просмотреть результаты. Обратный вызов после операции получает следующую структуру:

```
typedef struct _OB_POST_OPERATION_INFORMATION {
    _In_ OB_OPERATION Operation;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG KernelHandle:1;
            _In_ ULONG Reserved:31;
        };
    };
    _In_ PVOID       Object;
    _In_ POBJECT_TYPE ObjectType;
    _In_ PVOID CallContext;
    _In_ NTSTATUS ReturnStatus;
    _In_ POB_POST_OPERATION_PARAMETERS Parameters;
} OB_POST_OPERATION_INFORMATION,*POB_POST_OPERATION_INFORMATION;
```

Все это похоже на информацию обратного вызова перед операцией, за исключением следующего:

- Итоговый статус операции возвращается в `ReturnStatus`. В случае успешного выполнения клиент получает обратно действительный дескриптор (возможно, с сокращенной маской доступа).
- Объединение `Parameters` содержит всего один вид информации: маску доступа, предоставляемую клиенту (при условии статуса успешного выполнения).

Драйвер Process Protector

В драйвере Process Protector продемонстрировано использование обратных вызовов объектов. Он предназначен для защиты некоторых процессов от завершения, для чего он исключает маску доступа `PROCESS_TERMINATE` в запросах любых клиентов, пытающихся завершить эти «защищенные» процессы.

> Полный код проектов драйвера и клиента доступен в репозитории Github: https://github.com/zodiacon/windowskernelprogrammingbook.

Драйвер должен вести список защищенных процессов. В этом драйвере используется простой ограниченный массив для хранения идентификаторов процессов, находящихся под защитой драйвера. Для хранения глобальных данных драйвера используется следующая структура (определяемая в ProcessProtect.h):

```
#define DRIVER_PREFIX "ProcessProtect: "

#define PROCESS_TERMINATE 1

#include "FastMutex.h"
```

```
const int MaxPids = 256;

struct Globals {
    int PidsCount;       // Счетчик защищенных процессов
    ULONG Pids[MaxPids]; // PID защищенных процессов
    FastMutex Lock;
    PVOID RegHandle;     // Специальное значение для регистрации объектов

    void Init() {
        Lock.Init();
    }
};
```

Обратите внимание: значение PROCESS_TERMINATE необходимо объявить явно, так как оно не определяется в заголовках WDK (определяется только PROCESS_ALL_ACCESS). Его определение можно без особого труда найти в заголовках пользовательского режима или в документации.

В основном файле (ProcessProtect.cpp) объявляется глобальная переменная типа Globals с именем g_Data (и вызывается Init в начале DriverEntry).

Регистрация уведомлений объектов

Функция DriverEntry для драйвера Process Protector должна включать регистрацию обратных вызовов объектов для процессов. Сначала подготовим структуры для регистрации:

```
OB_OPERATION_REGISTRATION operations[] = {
    {
        PsProcessType, // Тип объекта
        OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE,
        OnPreOpenProcess, nullptr // Перед и после
    }
};

OB_CALLBACK_REGISTRATION reg = {
    OB_FLT_REGISTRATION_VERSION,
    1,   // счетчик операций
    RTL_CONSTANT_STRING(L"12345.6171"), // Altitude
    nullptr,       // контекст
    operations
};
```

Регистрация относится только к объектам процессов, предоставляется функция обратного вызова перед операцией. Функция должна исключать режим PROCESS_TERMINATE из маски доступа, запрашиваемой любым клиентом.

Теперь все готово для непосредственного выполнения регистрации:

```
do {
    status = ObRegisterCallbacks(&reg, &g_Data.RegHandle);
    if (!NT_SUCCESS(status)) {
        break;
    }
```

Управление защищенными процессами

Драйвер поддерживает массив идентификаторов процессов, находящихся под его защитой. Драйвер предоставляет три кода управляющих операций ввода/вывода для добавления и исключения PID, а также для очистки всего списка. Коды управляющих операций определяются в ProcessProtectCommon.h:

```
#define PROCESS_PROTECT_NAME L"ProcessProtect"

#define IOCTL_PROCESS_PROTECT_BY_PID \
    CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROCESS_UNPROTECT_BY_PID \
    CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROCESS_PROTECT_CLEAR \
    CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Для установления и снятия защиты процессов обработчик `IRP_MJ_DEVICE_CONTROL` получает массив идентификаторов PID (не обязательно один). Общая схема кода обработчика представляет собой стандартную конструкцию `switch` для известных кодов управляющих операций:

```
NTSTATUS ProcessProtectDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto status = STATUS_SUCCESS;
    auto len = 0;

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
        case IOCTL_PROCESS_PROTECT_BY_PID:
            //...
            break;

        case IOCTL_PROCESS_UNPROTECT_BY_PID:
            //...
            break;

        case IOCTL_PROCESS_PROTECT_CLEAR:
            //...
            break;

        default:
            status = STATUS_INVALID_DEVICE_REQUEST;
        break;
```

```
    }

    // Завершение запроса
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = len;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

Чтобы упростить добавление и удаление PID, мы создадим две вспомогательные функции:

```
bool AddProcess(ULONG pid) {
    for(int i = 0; i < MaxPids; i++)
        if (g_Data.Pids[i] == 0) {
            // Пустой слот
            g_Data.Pids[i] = pid;
            g_Data.PidsCount++;
            return true;
        }
    return false;
}

bool RemoveProcess(ULONG pid) {
    for (int i = 0; i < MaxPids; i++)
        if (g_Data.Pids[i] == pid) {
            g_Data.Pids[i] = 0;
            g_Data.PidsCount--;
            return true;
        }
    return false;
}
```

Обратите внимание: быстрый мьютекс в этих функциях не захватывается; это означает, что вызывающая сторона должна захватить быстрый мьютекс перед вызовом `AddProcess` или `RemoveProcess`.

Последняя вспомогательная функция ищет идентификатор процесса в массиве и возвращает `true`, если он будет найден:

```
bool FindProcess(ULONG pid) {
    for (int i = 0; i < MaxPids; i++)
        if (g_Data.Pids[i] == pid)
            return true;
    return false;
}
```

Все готово к реализации кодов управляющих операций ввода/вывода. Для добавления процесса необходимо найти пустой слот в массиве идентификаторов процессов и сохранить запрашиваемый идентификатор PID; конечно, мы можем получить сразу несколько PID.

```
case IOCTL_PROCESS_PROTECT_BY_PID:
{
    auto size = stack->Parameters.DeviceIoControl.InputBufferLength;
    if (size % sizeof(ULONG) != 0) {
        status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }

    auto data = (ULONG*)Irp->AssociatedIrp.SystemBuffer;

    AutoLock locker(g_Data.Lock);

    for (int i = 0; i < size / sizeof(ULONG); i++) {
        auto pid = data[i];
        if (pid == 0) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        if (FindProcess(pid))
            continue;

        if (g_Data.PidsCount == MaxPids) {
            status = STATUS_TOO_MANY_CONTEXT_IDS;
            break;
        }

        if (!AddProcess(pid)) {
            status = STATUS_UNSUCCESSFUL;
            break;
        }

        len += sizeof(ULONG);
    }

    break;
}
```

Сначала код проверяет размер буфера, который должен быть кратен 4 байтам (PID) и отличен от нуля. Затем он получает указатель на системный буфер (при этом используется буферизованный ввод/вывод `METHOD_BUFFERED` — см. главу 7). Далее захватывается быстрый мьютекс, после чего начинается цикл.

Цикл перебирает все PID, предоставленные в запросе, и если все следующие условия истинны, добавляет PID в массив:

- Значение PID не равно нулю (это значение PID всегда недопустимо, так как оно зарезервировано для процесса `Idle`).

- Значение PID не находится в массиве (`FindProcess` проверяет это условие).
- Количество управляемых PID не превысило `MaxPids`.

Удаление PID выполняется аналогичным образом. Необходимо сначала найти PID, а затем «удалить» его, поместив 0 в этот слот (задача решается функцией `RemoveProcess`):

```
case IOCTL_PROCESS_UNPROTECT_BY_PID:
{
    auto size = stack->Parameters.DeviceIoControl.InputBufferLength;
    if (size % sizeof(ULONG) != 0) {
        status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }

    auto data = (ULONG*)Irp->AssociatedIrp.SystemBuffer;

    AutoLock locker(g_Data.Lock);

    for (int i = 0; i < size / sizeof(ULONG); i++) {
        auto pid = data[i];
        if (pid == 0) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        if (!RemoveProcess(pid))
            continue;

        len += sizeof(ULONG);

        if (g_Data.PidsCount == 0)
            break;
    }

    break;
}
```

Не забудьте: использование `AutoLock` без параметризованного типа требует назначения стандарта C++ 17 в настройках C++ проекта.

Наконец, очистка списка выполняется достаточно просто, так как все происходит при удержании блокировки:

```
case IOCTL_PROCESS_PROTECT_CLEAR:
{
    AutoLock locker(g_Data.Lock);
    ::memset(&g_Data.Pids, 0, sizeof(g_Data.Pids));
    g_Data.PidsCount = 0;
    break;
}
```

Обратный вызов перед операцией

Самая важная часть драйвера — исключение маски `PROCESS_TERMINATE` для идентификаторов PID, в настоящее время защищенных от завершения:

```
OB_PREOP_CALLBACK_STATUS
OnPreOpenProcess(PVOID, POB_PRE_OPERATION_INFORMATION Info) {
    if(Info->KernelHandle)
        return OB_PREOP_SUCCESS;

    auto process = (PEPROCESS)Info->Object;
    auto pid = HandleToULong(PsGetProcessId(process));

    AutoLock locker(g_Data.Lock);
    if (FindProcess(pid)) {
        // Присутствует в списке, удалить маску завершения
        Info->Parameters->CreateHandleInformation.DesiredAccess &=
            ~PROCESS_TERMINATE;
    }

    return OB_PREOP_SUCCESS;
}
```

Если дескриптор является дескриптором режима ядра, мы разрешаем нормальное продолжение операции. И это логично, потому что драйвер не должен препятствовать нормальной работе кода режима ядра.

Затем нужно узнать идентификатор процесса, для которого открывается дескриптор. Данные передаются функции обратного вызова в виде указателя на объект. К счастью, PID легко определяется функцией API `PsGetProcessId`. Функция получает структуру `PEPROCESS` и возвращает идентификатор процесса.

Остается сделать последний шаг: проверить, защищен этот конкретный процесс или нет. Для этого вызывается функция `FindProcess` под защитой блокировки. Если процесс будет найден, то маска доступа `PROCESS_TERMINATE` исключается.

Клиентское приложение

Клиентское приложение должно уметь добавлять, исключать и очищать список процессов, выдавая правильные вызовы `DeviceIoControl`. Следующие команды демонстрируют интерфейс командной строки (предполагается, что исполняемому файлу присвоено имя Protect.exe):

`Protect.exe add 1200 2820` (защита устанавливается для PID 1200 и 2820)

`Protect.exe remove 2820` (защита снимается для PID 2820)

`Protect.exe clear` (все PID исключаются из системы защиты)

Функция `main` выглядит так:

```
int wmain(int argc, const wchar_t* argv[]) {
    if(argc < 2)
        return PrintUsage();

    enum class Options {
        Unknown,
        Add, Remove, Clear
    };
    Options option;
    if (::_wcsicmp(argv[1], L"add") == 0)
        option = Options::Add;
    else if (::_wcsicmp(argv[1], L"remove") == 0)
        option = Options::Remove;
    else if (::_wcsicmp(argv[1], L"clear") == 0)
        option = Options::Clear;
    else {
        printf("Unknown option.\n");
        return PrintUsage();
    }

    HANDLE hFile = ::CreateFile(L"\\\\.\\" PROCESS_PROTECT_NAME,
        GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return Error("Failed to open device");

    std::vector<DWORD> pids;
    BOOL success = FALSE;
    DWORD bytes;
    switch (option) {
        case Options::Add:
            pids = ParsePids(argv + 2, argc - 2);
            success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_BY_PID,
                pids.data(), static_cast<DWORD>(pids.size()) * sizeof(DWORD),
                nullptr, 0, &bytes, nullptr);
            break;

        case Options::Remove:
            pids = ParsePids(argv + 2, argc - 2);
            success = ::DeviceIoControl(hFile, IOCTL_PROCESS_UNPROTECT_BY_PID,
                pids.data(), static_cast<DWORD>(pids.size()) * sizeof(DWORD),
                nullptr, 0, &bytes, nullptr);
            break;

        case Options::Clear:
            success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_CLEAR,
                nullptr, 0, nullptr, 0, &bytes, nullptr);
            break;

    }

    if (!success)
        return Error("Failed in DeviceIoControl");

    printf("Operation succeeded.\n");
```

```
    ::CloseHandle(hFile);

    return 0;
}
```

Вспомогательная функция `ParsePids` разбирает идентификаторы процессов и возвращает их в форме `std::vector<DWORD>`, которая легко передается в виде массива при помощи метода `data()` для `std::vector<T>`:

```
std::vector<DWORD> ParsePids(const wchar_t* buffer[], int count) {
    std::vector<DWORD> pids;
    for (int i = 0; i < count; i++)
        pids.push_back(::_wtoi(buffer[i]));
    return pids;
}
```

Наконец, функция `Error` не отличается от использованной в предыдущих проектах, а функция `PrintUsage` просто выводит простую информацию об использовании.

Драйвер устанавливается и запускается как обычно:

```
sc create protect type= kernel binPath= c:\book\processprotect.sys

sc start protect
```

Чтобы протестировать драйвер, попробуйте запустить процесс (например, Notepad.exe), включите защиту, а потом попытайтесь уничтожить его из диспетчера задач. На рис. 9.1 показан работающий экземпляр Notepad.exe.

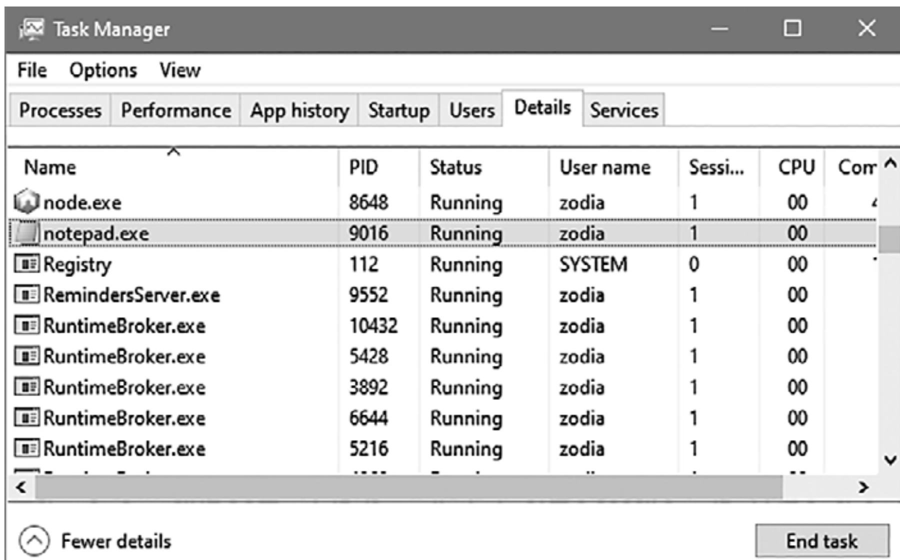


Рис. 9.1. Выполняемый экземпляр Notepad.exe

Включите для него защиту:

```
protect add 9016
```

Щелкните на кнопке Завершить процесс в диспетчере задач. На экране появляется сообщение об ошибке (рис. 9.2).

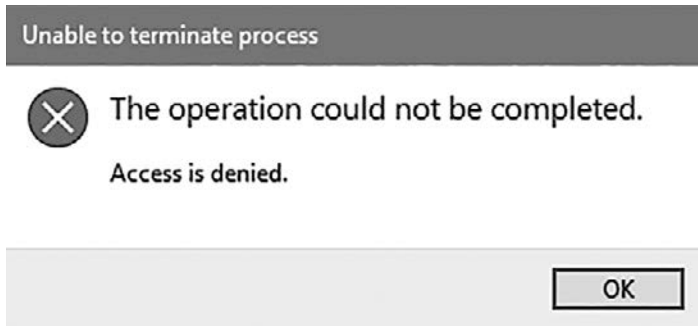


Рис. 9.2. Попытка завершения процесса

Снимите защиту и повторите попытку. На этот раз процесс завершается как обычно.

```
protect remove 9016
```

Именно с программой Блокнот даже при включенной защите щелчок на кнопке закрытия окна или выбор команды Файл ▸ Выход завершает процесс. Дело в том, что закрытие выполняется внутренним вызовом `ExitProcess`, в котором не задействованы никакие дескрипторы. А это означает, что разработанный нами механизм защиты хорошо подходит для процессов, не имеющих пользовательского интерфейса.

Добавьте код управляющей операции для запроса информации о процессах, выбранных в настоящий момент. Коду управляющей операции можно присвоить имя `IOCTL_PROCESS_QUERY_PIDS`.

Уведомления реестра

Диспетчер конфигурации (часть исполнительной системы, обеспечивающая работу с реестром) может использоваться для регистрации уведомлений об обращениях к разделам реестра.

Регистрация на эти уведомления осуществляется функцией API CmRegisterCallbackEx. Ее прототип выглядит так:

```
NTSTATUS CmRegisterCallbackEx (
    _In_ PEX_CALLBACK_FUNCTION   Function,
    _In_ PCUNICODE_STRING        Altitude,
    _In_ PVOID Driver,           // PDRIVER_OBJECT
    _In_opt_ PVOID               Context,
    _Out_ PLARGE_INTEGER         Cookie,
    _Reserved_ PVOID             Reserved
```

Function — сама функция обратного вызова, которую мы вскоре рассмотрим более подробно. Altitude — приоритет обратных вызовов драйвера, по смыслу практически идентичный приоритету уведомлений объектов. В аргументе Driver должен содержаться объект драйвера, доступный в DriverEntry. Context, — определяемое драйвером значение, которое передается функции обратного вызова в неизменном виде. Наконец, в Cookie хранится результат регистрации, если она прошла успешно. Это специальное значение должно быть передано CmUnregisterCallback для отмены регистрации.

> Немного странно, что разные API регистрации ведут себя непоследовательно в отношении регистрации/отмены регистрации: CmRegisterCallbackEx возвращает LARGE_INTEGER для представления результата регистрации; ObRegisterCallbacks возвращает PVOID; функции регистрации процессов и потоков не возвращают ничего (во внутренней реализации для идентификации регистрации используется адрес самой функции обратного вызова). Наконец, отмена регистрации процессов и потоков выполняется асимметричными функциями API. Ну что ж!

Функция обратного вызова выглядит вполне привычно:

```
NTSTATUS RegistryCallback (
    _In_ PVOID CallbackContext,
    _In_opt_ PVOID Argument1,
    _In_opt_ PVOID Argument2);
```

CallbackContext — аргумент Context, передаваемый CmRegisterCallbackEx. Первый обобщенный аргумент в действительности представляет собой перечисление REG_NOTIFY_CLASS, описывающее операцию, для которой активизируется обратный вызов, а также разновидность уведомления (перед или после операции). Второй аргумент содержит указатель на конкретную структуру для уведомлений этого типа. Драйвер обычно содержит конструкцию switch по типу уведомления:

```
NTSTATUS OnRegistryNotify(PVOID, PVOID Argument1, PVOID Argument2) {
    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)Argument1) {
        //...
    }
```

В табл. 9.1 приведены некоторые значения из перечисления REG_NOTIFY_CLASS и соответствующие структуры, передаваемые в Argument2.

Таблица 9.1. Некоторые уведомления реестра и соответствующие структуры

Уведомление	**Соответствующая структура**
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtPreSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPostSetValueKey	REG_POST_OPERATION_INFORMATION
RegNtPreCreateKey	REG_PRE_CREATE_KEY_INFORMATION
RegNtPostCreateKey	REG_POST_CREATE_KEY_INFORMATION

Из табл. 9.1 видно, что уведомления после операции используют одну и ту же структуру REG_POST_CREATE_KEY_INFORMATION.

Обработка уведомлений перед операцией

Для уведомлений перед операцией функция обратного вызова активизируется перед их выполнением диспетчером конфигурации.

У драйвера имеются следующие варианты:

- Вернуть STATUS_SUCCESS из обратного вызова, приказывая диспетчеру конфигурации продолжить нормальное выполнение операции.
- Вернуть из обратного вызова некоторый статус ошибки. В этом случае диспетчер конфигурации возвращает управление на сторону вызова с этим статусом, и обратный вызов после операции не активизируется.
- Как-то обработать запрос, а затем вернуть STATUS_CALLBACK_BYPASS из обратного вызова. Диспетчер конфигурации возвращает признак успеха на сторону вызова и не активизирует обратный вызов после операции. Драйвер должен обеспечить присутствие правильных значений в структуре REG_xxx_KEY_INFORMATION, предоставляемой функции обратного вызова.

Обработка уведомлений после операции

После того как операция будет завершена (а драйвер не запретил уведомление после операции), обратный вызов активизируется диспетчером конфигурации после выполнения операции. Структура, предоставляемая для уведомлений после операции, выглядит так:

```
typedef struct _REG_POST_OPERATION_INFORMATION {
    PVOID Object;          // Входные данные
    NTSTATUS Status;       // Входные данные
    PVOID PreInformation;  // Предварительная информация
    NTSTATUS ReturnStatus; // Обратный вызов может изменить результат операции
    PVOID CallContext;
    PVOID ObjectContext;
    PVOID Reserved;
} REG_POST_OPERATION_INFORMATION,*PREG_POST_OPERATION_INFORMATION;
```

У обратного вызова для уведомлений после операции имеются следующие варианты:

- Проверить результат реализации и сделать что-то полезное (например, сохранить в журнале).
- Изменить возвращаемый статус, присвоив новое значение полю ReturnStatus структуры уведомления после операции, после чего вернуть STATUS_CALLBACK_BYPASS. Диспетчер конфигурации возвращает новый статус на сторону вызова.
- Изменить выходные параметры в структуре REG_xxx_KEY_INFORMATION и вернуть STATUS_SUCCESS. Диспетчер конфигурации возвращает новые данные на сторону вызова.

Поле PreInformation структуры уведомления после операции содержит указатель на структуру уведомления перед операцией.

Факторы быстродействия

Обратный вызов реестра активизируется для *каждой* операции с реестром; не существует априорного способа отфильтровать операции нужного типа. Это означает, что обратный вызов должен отработать как можно быстрее, потому что вызывающая сторона ожидает. Кроме того, в цепочке обратных вызовов могут находиться сразу несколько драйверов.

Некоторые операции с реестром, особенно операции чтения, происходят в большом количестве, поэтому драйверу следует по возможности избегать обработки операций чтения. Если же без обработки операций чтения не обойтись, следует как минимум ограничиться разделами, представляющими интерес, например содержимым HKLM\System\CurrentControlSet (приведено просто для примера).

Операции чтения и создания используются намного реже, поэтому в таких случаях драйвер при необходимости может выполнять больший объем работы.

Вывод прост: нужно делать как можно меньше с минимальным количеством разделов.

Реализация уведомлений реестра

Расширим драйвер SysMon из главы 8, чтобы он включал уведомления для некоторых операций с реестром. Для примера добавим уведомления для записи в иерархии раздела `HKEY_LOCAL_MACHINE`.

Начнем с определения структуры данных для передаваемой информации (в SysMonCommon.h):

```
struct RegistrySetValueInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ThreadId;
    WCHAR KeyName[256];  // Полное имя раздела
    WCHAR ValueName[64]; // Имя параметра
    ULONG DataType;      // REG_xxx
    UCHAR Data[128];     // Данные
    ULONG DataSize;      // Размер данных
};
```

Ради простоты для передачи информации будут использоваться массивы фиксированного размера. В драйвере коммерческого уровня массив лучше сделать динамическим, чтобы сэкономить память и предоставить более полную информацию там, где потребуется.

Массив `Data` содержит непосредственно записанные данные. Естественно, массив нужно каким-то образом ограничить, так как его размер может стать практически неограниченным.

`DataType` — одна из констант семейства `REG_xxx`: `REG_SZ`, `REG_DWORD`, `REG_BINARY` и т. д. Эти значения одинаковы как в пользовательском режиме, так и в режиме ядра.

Затем добавим новый тип события для этого уведомления:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit,
    ImageLoad,
    RegistrySetValue // Новое значение
};
```

В DriverEntry необходимо добавить регистрацию обратного вызова для уведомлений реестра в блок do/while(false). Возвращаемое специальное значение, представляющее регистрацию, хранится в структуре Globals:

```
UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"7657.124");
status = CmRegisterCallbackEx(OnRegistryNotify, &altitude, DriverObject,
    nullptr, &g_Globals.RegCookie, nullptr);
if(!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set registry callback (%08X)\n",
        status));
    break;
}
```

Конечно, вы должны отменить регистрацию уведомлений в функции выгрузки:

```
CmUnRegisterCallback(g_Globals.RegCookie);
```

Обработка обратных вызовов уведомлений реестра

Наша функция обратного вызова должна обрабатывать только попытки записи в HKEY_LOCAL_MACHINE. Сначала выбирается операция, представляющая интерес:

```
NTSTATUS OnRegistryNotify(PVOID context, PVOID arg1, PVOID arg2) {
    UNREFERENCED_PARAMETER(context);

    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)arg1) {
        case RegNtPostSetValueKey:
        //...
    }
    return STATUS_SUCCESS;
}
```

В этом драйвере другие операции нас не интересуют, поэтому после switch просто возвращается статус успешного выполнения. Обратите внимание: проверяется уведомление после операции, так как для драйвера представляет интерес только результат. Затем в нужной секции case второй аргумент преобразует данные уведомления после операции, и мы проверяем, успешно ли завершилась операция:

```
auto args = (REG_POST_OPERATION_INFORMATION*)arg2;
if (!NT_SUCCESS(args->Status))
    break;
```

Если операция не была успешной, выполнение прерывается. В нашем драйвере это решение было выбрано произвольно; для другого драйвера неудачные попытки могут представлять интерес для дальнейшего анализа.

Затем необходимо проверить, находится ли раздел в ветви HKLM. Если нет, его можно пропустить. Внутренние пути реестра в представлении ядра всегда на-

чинаются с корневого раздела \REGISTRY\. Затем идет раздел MACHINE\ для куста локальной машины — то же, что HKEY_LOCAL_MACHINE в коде пользовательского режима. Следовательно, необходимо проверить, начинается ли раздел с префикса \REGISTRY\MACHINE\.

Путь к разделу не хранится ни в структуре данных после операции, ни даже в структуре данных перед операцией. Вместо этого сам объект раздела реестра передается в составе информационной структуры после операции. Затем необходимо извлечь имя раздела функцией CmCallbackGetKeyObjectIDEx и проверить, начинается ли он с префикса \REGISTRY\MACHINE\:

```
static const WCHAR machine[] = L"\\REGISTRY\\MACHINE\\";

PCUNICODE_STRING name;
if (NT_SUCCESS(CmCallbackGetKeyObjectIDEx(&g_Globals.RegCookie, args->Object,
    nullptr, &name, 0))) {
    // Попытки записи в другие ветви, кроме HKLM, отфильтровываются
    if (::wcsncmp(name->Buffer, machine, ARRAYSIZE(machine) - 1) == 0) {
```

Если условие выполняется, необходимо сохранить информацию об операции в структуре уведомления и занести ее в очередь. Эта информация (тип данных, имя параметра, фактическое значение и т. д.) предоставляется с информацией уведомления перед операцией, которая, к счастью, доступна в составе непосредственно полученной структуры уведомления после операции.

```
auto preInfo = (REG_SET_VALUE_KEY_INFORMATION*)args->PreInformation;
NT_ASSERT(preInfo);

auto size = sizeof(FullItem<RegistrySetValueInfo>);
auto info = (FullItem<RegistrySetValueInfo>*)ExAllocatePoolWithTag(PagedPool,
    size, DRIVER_TAG);
if (info == nullptr)
    break;

// Структура обнуляется, чтобы строки завершались нулем при копировании
RtlZeroMemory(info, size);

// Заполнение стандартных данных
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Size = sizeof(item);
item.Type = ItemType::RegistrySetValue;

// Получение клиентского PID/TID (вызывающая сторона)
item.ProcessId = HandleToULong(PsGetCurrentProcessId());
item.ThreadId = HandleToULong(PsGetCurrentThreadId());

// Получение информации о конкретном разделе/параметре
::wcsncpy_s(item.KeyName, name->Buffer, name->Length / sizeof(WCHAR) - 1);
::wcsncpy_s(item.ValueName, preInfo->ValueName->Buffer,
    preInfo->ValueName->Length / sizeof(WCHAR) - 1);
```

```
item.DataType = preInfo->Type;
item.DataSize = preInfo->DataSize;
::memcpy(item.Data, preInfo->Data, min(item.DataSize, sizeof(item.Data)));

PushItem(&info->Entry);
```

Конкретная структура уведомления перед операцией (`REG_SET_VALUE_KEY_INFORMATION`) содержит искомую информацию. Код принимает меры к тому, чтобы предотвратить излишнее копирование с переполнением статически выделенных буферов.

Наконец, если вызов `CmCallbackGetKeyObjectIDEx` завершился успехом, имя полученного ключа необходимо освободить явно:

```
CmCallbackReleaseKeyObjectIDEx(name);
```

Обновленный код клиента

Клиентское приложение необходимо изменить для поддержки нового типа события. Одна из возможных реализаций:

```
case ItemType::RegistrySetValue:
{
    DisplayTime(header->Time);
    auto info = (RegistrySetValueInfo*)buffer;
    printf("Registry write PID=%d: %ws\\%ws type: %d size: %d data: ",
        info->ProcessId, info->KeyName, info->ValueName,
        info->DataType, info->DataSize);

    switch (info->DataType) {
        case REG_DWORD:
            printf("0x%08X\n", *(DWORD*)info->Data);
            break;

        case REG_SZ:
        case REG_EXPAND_SZ:
            printf("%ws\n", (WCHAR*)info->Data);
            break;

        case REG_BINARY:
            DisplayBinary(info->Data, min(info->DataSize, sizeof(info->Data)));
            break;

        // Другие случаи... (REG_QWORD, REG_LINK и т. д.)
        default:
            DisplayBinary(info->Data, min(info->DataSize, sizeof(info->Data)));
            break;
    }
    break;
}
```

`DisplayBinary` — простая вспомогательная функция, которая выводит двоичные данные в виде серии шестнадцатеричных значений. Приведу ее код для полноты:

```
void DisplayBinary(const UCHAR* buffer, DWORD size) {
    for (DWORD i = 0; i < size; i++)
        printf("%02X ", buffer[i]);
    printf("\n");
}
```

Пример вывода расширенного клиента и драйвера:

```
19:22:21.509: Thread 6488 Exited from process 8808
19:22:21.509: Thread 5348 Created in process 8252
19:22:21.510: Thread 5348 Exited from process 8252
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                                FileAttributes\
FilteredOut type: 4 size: 4 data: 0x00000000
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                               UsnSourceFilte\
redOut type: 4 size: 4 data: 0x00000008
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                               UsnReasonFilte\
redOut type: 4 size: 4 data: 0x00000000
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                               ConfigFlags ty\
pe: 4 size: 4 data: 0x00000001
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                               ScopeToMonitor\
 type: 1 size: 270 data: C:\Users\zodia\AppData\Local\Packages\
                            Microsoft.Windows.Con\
tentD19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\
                                             Microsoft\Win\
dows Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                                    Monitored\
PathRegularExpressionExclusion type: 1 size: 2 data:
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
                               ApplicationNam\
e type: 1 size: 36 data: RuntimeBroker.exe
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
                                        Windows \
```

```
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
ClientId type: 4 size: 4 data: 0x00000001
19:22:21.531: Registry write PID=7288: \REGISTRY\MACHINE\SOFTWARE\Microsoft\
Windows \
Search\FileChangeClientConfigs\{5B8A4E77-3A02-4093-BDDC-B46FAB03AEF5}\
VolumeIndex type: 4 size: 4 data: 0x00000001
19:22:21.678: Thread 4680 Exited from process 6040
19:22:21.678: Thread 4760 Exited from process 6040
```

Расширьте SysMon и добавьте коды управляющих операций ввода/вывода для разрешения/запрета некоторых типов уведомлений (процессы, потоки, загрузка образов, реестр).

Упражнения

1. Реализуйте драйвер, который не допускает внедрения потоков в другие процессы, если только целевой процесс не находится под управлением отладчика.
2. Реализуйте драйвер, который защищает раздел реестра от изменений. Клиент может передавать драйверу разделы реестра, для которых устанавливается или снимается защита.
3. Реализуйте драйвер, который перенаправляет операции записи в реестр от выбранных процессов (настраиваемых в клиентском приложении) в свой приватный раздел, если процессы обращаются к HKEY_LOCAL_MACHINE. Если приложение записывает данные, они отправляются в приватное хранилище. Если приложение читает данные, сначала следует проверить приватное хранилище, а если значение там не обнаружено, обратиться к реальному разделу реестра. Такой механизм хранения может рассматриваться как один из аспектов защитной изоляции (sandboxing) приложений.

Итоги

В этой главе рассматривались два механизма обратного вызова, поддерживаемых ядром: уведомления о получении дескрипторов некоторых объектов и обращениях к реестру. В следующей главе мы займемся изучением совершенно новой области — мини-фильтров файловой системы.

Глава 10

Мини-фильтры файловой системы

Операции ввода/вывода с файлами выполняются в файловой системе. Windows поддерживает несколько разных файловых систем, прежде всего NTFS — ее «родную» файловую систему. Механизм *фильтров файловой системы* используется драйверами для перехвата вызовов, обращенных к файловой системе. Он приносит пользу во многих видах программных продуктов: антивирусов, систем резервного копирования, средств шифрования и т. д.

В системе Windows в течение долгого времени поддерживалась модель фильтрации, называемая фильтрами файловой системы; сейчас она обозначается термином «наследные фильтры файловой системы». На смену механизму наследных фильтров была разработана более новая модель — *мини-фильтры файловой системы.* Мини-фильтры проще пишутся во многих отношениях и в настоящее время считаются предпочтительным механизмом разработки фильтрующих драйверов файловой системы. В этой главе рассматриваются основы мини-фильтров файловой системы.

В этой главе:

- Введение
- Загрузка и выгрузка
- Инициализация
- Установка
- Обработка операций ввода/вывода
- Драйвер Delete Protector
- Имена файлов
- Контексты
- Инициирование запросов ввода/вывода
- File Backup Driver
- Взаимодействие с пользовательским режимом
- Отладка
- Упражнения

Введение

Наследные фильтры файловой системы пользовались дурной славой из-за сложностей их написания. Разработчик драйвера должен был учитывать множество мелких подробностей, многие из которых реализовывались шаблонным кодом, а это усложняло разработку. Наследные фильтры не могли выгружаться во время работы системы — это означало, что для загрузки обновленной версии драйвера приходилось перезагружать систему. Модель мини-фильтров делает возможной динамическую загрузку и выгрузку драйверов, что существенно упрощает процесс разработки.

Во внутренней реализации мини-фильтрами управляет наследный фильтр, называемый диспетчером фильтров (Filter Manager). Типичная иерархия фильтров показана на рис. 10.1.

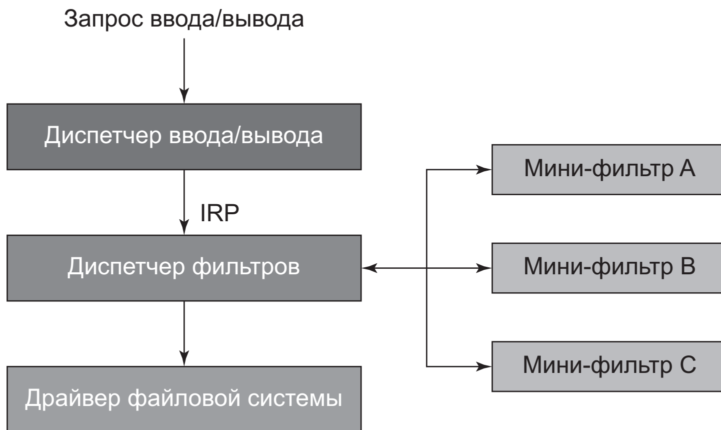


Рис. 10.1. Мини-фильтры под управлением диспетчера фильтров

Каждый мини-фильтр имеет собственный параметр `Altitude`, который определяет его относительную позицию в стеке устройства. Диспетчер фильтров получает IRP, как и любые другие наследные фильтры, а затем обращается с вызовами к мини-фильтрам, находящимся под его управлением, в порядке убывания `Altitude`.

В некоторых нетипичных ситуациях в иерархии может присутствовать еще один наследный фильтр; возникает «расщепление» мини-фильтров: одни находятся выше наследного драйвера, другие — ниже. В таком случае загружаются несколько экземпляров диспетчера фильтров, каждый из которых управляет собственными мини-фильтрами. Каждый экземпляр диспетчера фильтров называется *кадром* (frame). На рис. 10.2 показан пример с двумя кадрами.

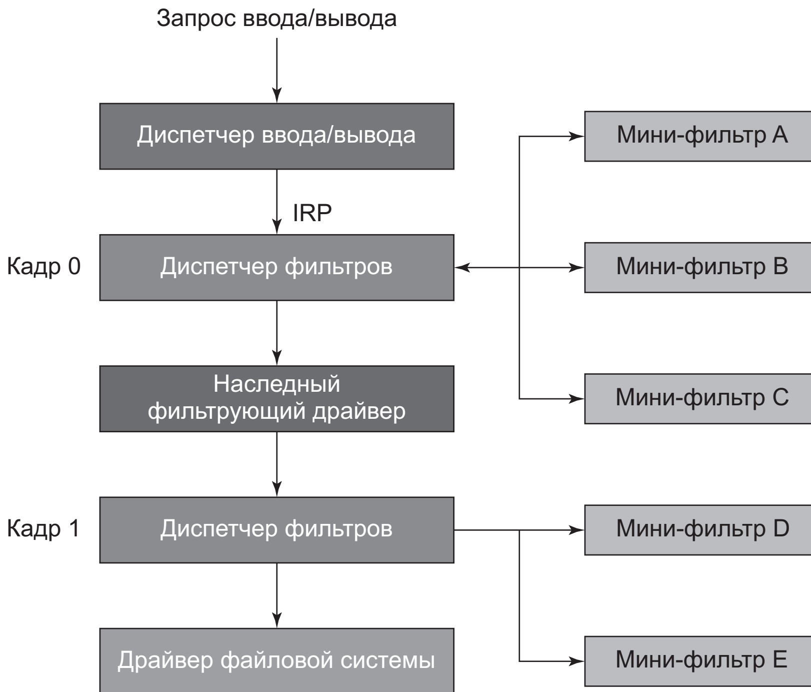


Рис. 10.2. Мини-фильтры в двух кадрах диспетчера фильтров

Загрузка и выгрузка

Драйверы мини-фильтров могут загружаться точно так же, как любые другие драйверы. В пользовательском режиме для этого используется функция `FilterLoad`, которой передается имя драйвера (его раздел в реестре `HKLM\System\CurrentControlSet\Services\`*имя_драйвера*). Во внутренней реализации вызывается функция API режима ядра `FltLoadFilter` с такой же семантикой. Как и в случае с любым другим драйвером, при вызове из пользовательского режима функция `SeLoadDriverPrivilege` должна быть включена в маркер вызывающей стороны. По умолчанию она присутствует в маркерах административного уровня, но не в стандартных маркерах пользователей.

> Загрузка драйвера мини-фильтра эквивалентна загрузке стандартного программного драйвера. К процессу выгрузки это не относится.

Выгрузка мини-фильтра осуществляется функцией API `FilterUnload` в пользовательском режиме или функцией `FltUnloadFilter` в режиме ядра. Для ее вы-

полнения необходим такой же уровень привилегий, как и для загрузки, однако успех не гарантирован, поскольку для мини-фильтра будет вызвана *функция обратного вызова выгрузки* фильтра (см. далее). Функция может завершить запрос неудачей, так что драйвер останется в системе.

Функции API для загрузки и выгрузки драйверов удобны, однако в процессе разработки проще воспользоваться встроенной программой fltmc.exe, которая делает все это (и не только). При вызове без списка аргументов (из окна командной строки с повышенными привилегиями) программа выводит список мини-фильтров, загруженных в настоящий момент. На моей машине с Windows 10 Pro версии 1903 он выглядит так:

```
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 указывается имя драйвера, текущее количество выполняемых экземпляров (каждый экземпляр присоединен к определенному тому), его приоритет `Altitude` и кадр диспетчера фильтров, частью которого он является.

Возможно, вас интересует, почему существуют драйверы с разным количеством экземпляров. В двух словах: потому, что драйвер сам решает, присоединяться к заданному тому или нет (эта тема более подробно рассматривается в этой главе).

При загрузке драйвера программой fltmc.exe используется ключ `load`:

```
fltmc load myfilter
```

Соответственно, при выгрузке в командную строку добавляется ключ:

```
fltmc unload myfilter
```

`fltmc` также поддерживает другие параметры. Чтобы вывести полный список, введите команду `fltmc -?`. Например, подробная информация обо всех

экземплярах для каждого драйвера выводится командой `fltmc instances`, а список всех томов, смонтированных в системе, выводится командой `fltmc volumes`. Позднее в этой главе будет показано, как передать эту информацию драйверу.

Драйверы и фильтры файловой системы создаются в каталоге `FileSystem` пространства имен диспетчера объектов. На рис. 10.3 показан этот каталог в программе WinObj.

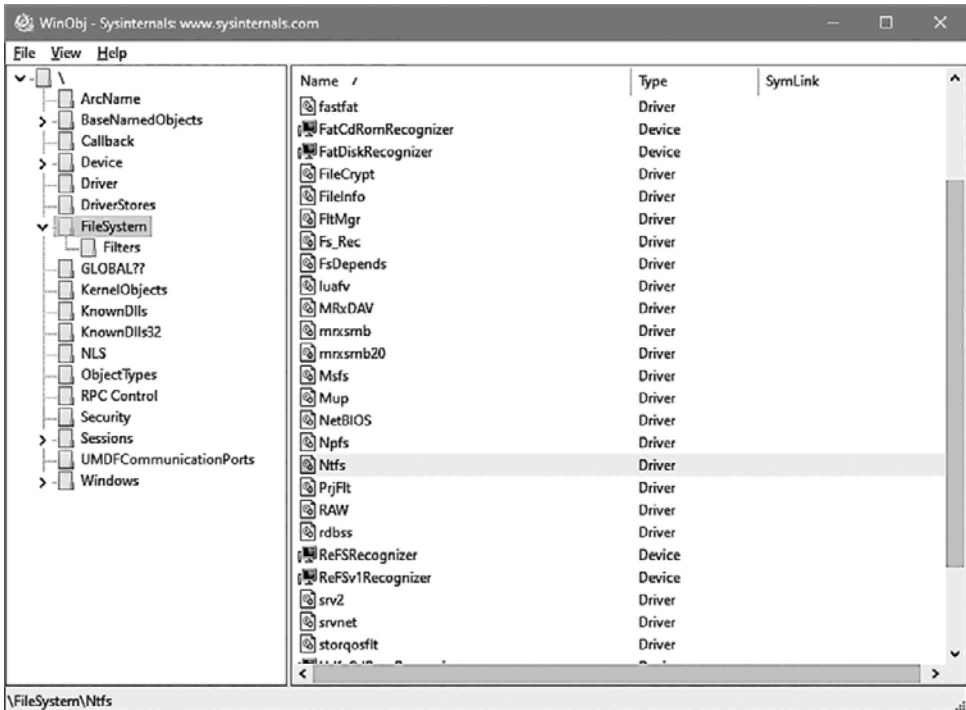


Рис. 10.3. Драйверы файловой системы, фильтры и мини-фильтры в WinObj

Инициализация

Драйвер мини-фильтра файловой системы содержит функцию `DriverEntry`, как и любой другой драйвер. Драйвер должен зарегистрироваться как мини-фильтр у диспетчера фильтров и задать различные настройки — например, операции, которые он желает перехватывать. Драйвер создает соответствующие структуры данных и вызывает функцию `FltRegisterFilter` для регистрации. В случае

успеха драйвер может выполнить дальнейшую необходимую инициализацию и вызвать `FltStartFiltering`, для того чтобы начать фильтрацию. Обратите внимание: драйверу не нужно создавать функции диспетчеризации (`IRP_MJ_READ`, `IRP_MJ_WRITE` и т. д.) самостоятельно. Дело в том, что драйвер не входит в путь ввода/вывода напрямую — в отличие от диспетчера фильтров.

Прототип функции `FltRegisterFilter` выглядит так:

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

Обязательная структура `FLT_REGISTRATION` предоставляет всю информацию, необходимую для регистрации. Ее определение:

```
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 инкапсулирован значительный объем информации. Ниже перечислены важнейшие поля:

- `Size` — в этом поле должен быть задан размер структуры, который может зависеть от целевой версии Windows (заданной в свойствах проекта). Драйверы обычно просто задают значение `sizeof(FLT_REGISTRATION)`.

- `Version` — версия, также основанная на целевой версии Windows. Драйверы используют `FLT_REGISTRATION_VERSION`.
- `Flags` — нуль или комбинация следующих значений:
  - `FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP` — драйвер не поддерживает запрос на остановку независимо от других параметров.
  - `FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS` — драйвер поддерживает именованные каналы и почтовые слоты и хочет фильтровать запросы к этим файловым системам (за дополнительной информацией обращайтесь к врезке «Каналы и почтовые слоты»).
  - `FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME` (Windows 10 версии 1607 и выше) — драйвер поддерживает присоединение к томам DAX (Direct Access Volume), если такой том доступен (см. далее врезку «DAX»).

КАНАЛЫ И ПОЧТОВЫЕ СЛОТЫ

Именованные каналы представляют собой одно- или двусторонние механизмы связи сервера с одним или несколькими клиентами, реализованные в виде файловой системы (npfs.sys). Windows API предоставляет специальные функции для создания канальных серверов и клиентов. Функция `CreateNamedPipe` создает сервер именованного канала, к которому клиенты могут подключаться обычной функцией `CreateFile` с «именем файла» в формате: `\\<сервер>\pipe\<канал>`.

Почтовые слоты — механизм односторонних коммуникаций, реализованный в виде файловой системы (msfs.sys). Серверный процесс открывает почтовый слот (своего рода почтовый ящик), в который клиенты могут отправлять сообщения. Функция `CreateMailslot` создает почтовый слот, к которому клиенты могут подключаться обычной функцией `CreateFile` с «именем файла» в формате: `\\<сервер>\mailslot\<почтовый_слот>`.

ТОМА ПРЯМОГО ДОСТУПА (DAX)

Тома прямого доступа (direct access volumes, DAX или DAS) — относительно новая возможность, добавленная в Windows 10 версии 1607 для поддержки новой разновидности хранения данных, базирующейся на прямом доступе к байтовым данным. Технология DAX поддерживается новым типом оборудования хранения данных Storage Class Memory — носителем долгосрочного хранения данных с производительностью, близкой к производительности ОЗУ (дополнительную информацию можно найти в интернете).

- `ContextRegistration` — необязательный указатель на массив структур `FLT_CONTEXT_REGISTRATION`; каждый элемент массива представляет контекст, который может использоваться драйвером в его работе. Под «контекстом» подразумеваются некие данные, определяемые драйвером, которые могут связываться с сущностями файловой системы (например, файлами и томами). Контексты будут более подробно рассмотрены позднее в этой главе. Некоторым драйверам контексты не нужны; в этом случае полю можно присвоить `NULL`.
- `OperationRegistration` — самое важное поле. Содержит указатель на массив структур `FLT_OPERATION_REGISTRATION`, каждая из которых определяет операцию и признак того, в какой момент драйвер хочет активизировать обратный вызов — перед и/или после операции. Более подробное описание приведено в следующем разделе.
- `FilterUnloadCallback` — функция, которая должна вызываться перед выгрузкой драйвера. Если задано значение `NULL`, значит, драйвер не может быть выгружен. Если драйвер устанавливает обратный вызов и возвращает код успеха, то драйвер выгружается; в этом случае драйвер должен вызвать функцию `FltUnregisterFilter`, чтобы отменить регистрацию перед выгрузкой. При возвращении любого другого статуса, кроме успеха, драйвер не выгружается.
- `InstanceSetupCallback` — этот обратный вызов позволяет драйверу получать уведомления перед присоединением экземпляра к новому тому. Драйвер может вернуть `STATUS_SUCCESS` для присоединения или `STATUS_FLT_DO_NOT_ATTACH`, если драйвер хочет запретить присоединение к новому тому.
- `InstanceQueryTeardownCallback` — необязательная функция обратного вызова, активизируемая перед отсоединением от тома. Это может произойти из-за явного запроса на отсоединение функцией `FltDetachVolume` в режиме ядра или `FilterDetach` в пользовательском режиме. Если вместо этого обратный вызов задает `NULL`, операция отсоединения отменяется.
- `InstanceTeardownStartCallback` — необязательный обратный вызов, активизируемый при уничтожении экземпляра. Драйвер должен завершить все незавершенные операции, так чтобы уничтожение экземпляра можно было завершить. Передача `NULL` в этом необязательном обратном вызове не предотвращает уничтожение экземпляра (для предотвращения следует использовать предыдущий обратный вызов).
- `InstanceTeardownCompleteCallback` — необязательный обратный вызов, активизируемый после завершения или отмены всех незавершенных операций ввода/вывода.

Все остальные поля обратных вызовов необязательны и редко используются на практике. В книге они не рассматриваются.

Регистрация обратных вызовов

Мини-фильтр должен указать, какие операции представляют для него интерес. Эта информация предоставляется в момент регистрации мини-фильтра в массиве структур FLT_OPERATION_REGISTRATION, которые определяются следующим образом:

```
typedef struct _FLT_OPERATION_REGISTRATION {
    UCHAR MajorFunction;
    FLT_OPERATION_REGISTRATION_FLAGS Flags;
    PFLT_PRE_OPERATION_CALLBACK PreOperation;
    PFLT_POST_OPERATION_CALLBACK PostOperation;

    PVOID Reserved1;   // Зарезервировано
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

Сама операция описывается первичным кодом функции. Большинство этих кодов совпадают с теми, которые встречались нам в предыдущих главах: IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE и т. д. Тем не менее существуют и другие операции, описываемые первичной функцией, которые не имеют реальной функции диспетчеризации. Эта абстракция, предоставляемая диспетчером фильтров, помогает изолировать мини-фильтр от информации о конкретном источнике операции — это может быть реальный IRP-запрос или другая операция, абстрагированная в форме IRP. Кроме того, файловые системы поддерживают еще один механизм получения запросов, называемый «быстрым вводом/выводом». Быстрый ввод/вывод используется для синхронного ввода/вывода с кэшированными файлами. Запросы быстрого ввода/вывода инициируют прямую передачу данных между пользовательскими буферами и системным кэшем, в обход файловой системы и стека драйверов накопителей, предотвращающие излишние затраты. Канонический пример: быстрый ввод/вывод поддерживается драйвером файловой системы NTFS.

> Быстрый ввод/вывод инициализируется выделением памяти для структуры FAST_IO_DISPATCH (содержащей длинный список функций обратного вызова), заполнением списка и присваиванием этой структуры полю FastIoDispatch структуры DRIVER_OBJECT.

Эту информацию можно просмотреть в отладчике ядра командой !drvobj, как в следующем примере 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

(...)

[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

(...)

Processed 10 device objects.
```

Диспетчер фильтров абстрагирует операции ввода/вывода независимо от того, на чем они базируются на IRP или быстром вводе/выводе. Мини-фильтры могут перехватывать любые такие запросы. Например, если драйвер не заинтересован в быстром вводе/выводе, он может проверить фактический тип запроса, предоставленного диспетчером фильтров, при помощи макросов `FLT_IS_FASTIO_OPERATION` и/или `FLT_IS_IRP_OPERATION`.

В табл. 10.1 перечислены распространенные первичные функции для мини-фильтров файловой системы с краткими описаниями.

Таблица 10.1. Распространенные первичные функции

Первичная функция	Функция диспетчеризации?	Описание
IRP_MJ_CREATE	Да	Создание или открытие файла/каталога
IRP_MJ_READ	Да	Чтение из файла
IRP_MJ_WRITE	Да	Запись в файл
IRP_MJ_QUERY_EA	Да	Чтение расширенных атрибутов из файла/каталога
IRP_MJ_DIRECTORY_CONTROL	Да	Запрос к каталогу
IRP_MJ_FILE_SYSTEM_CONTROL	Да	Управляющий запрос ввода/вывода к устройству файловой системы
IRP_MJ_SET_INFORMATION	Да	Различные информационные настройки файла (удаление, переименование)
IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION	Нет	Открытие секции (файла, отображаемого в память)
IRP_MJ_OPERATION_END	Нет	Признак конца массива обратных вызовов операций

Второе поле `FLT_OPERATION_REGISTRATION` может содержать нуль или комбинацию следующих флагов, влияющих на операции чтения и записи:

- `FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO` — не активизировать обратный вызов(-ы) для кэшированного ввода/вывода (например, быстрых операций ввода/вывода, которые всегда кэшируются).
- `FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO` — не активизировать обратный вызов(-ы) для страничного ввода/вывода (только для операций на базе IRP).

- FLTFL_OPERATION_REGISTRATION_SKIP_NON_DASD_IO — не активизировать обратный вызов(-ы) для томов DAX.

Следующие два поля содержат обратные вызовы перед и после операции, из которых хотя бы один должен быть отличен от NULL (а иначе зачем бы они были нужны?). Пример инициализации массива структур FLT_OPERATION_REGISTRATION (для вымышленного драйвера «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 }
};
```

При наличии такого массива регистрация драйвера, не требующего никаких контекстов, может осуществляться следующим кодом:

```
const FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0,                             // Флаги
    nullptr,                       // Контекст
    Callbacks,                     // Обратные вызовы операций
    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;
    //...
    status = FltRegisterFilter(DriverObject, &FilterRegistration, &FilterHandle);
    if(NT_SUCCESS(status)) {
        // Непосредственное начало фильтрации ввода/вывода
        status = FltStartFiltering(FilterHandle);
        if(!NT_SUCCESS(status))
            FltUnregisterFilter(FilterHandle);
    }
    return status;
}
```

Приоритет Altitude

Как вы уже видели, мини-фильтры файловой системы должны обладать приоритетом `Altitude`, обозначающим их относительную «позицию» в иерархии фильтров файловой системы. В отличие от приоритетов `Altitude`, с которыми вы уже сталкивались при работе с обратными вызовами уведомлений объектов и реестра, значение `Altitude` у мини-фильтров может быть потенциально важным.

Во-первых, значение `Altitude` не передается как часть регистрации мини-фильтра, а читается из реестра. При установке драйвера его значение `Altitude` записывается в соответствующий раздел реестра. На рис. 10.4 изображена запись реестра для встроенного драйвера мини-фильтра Fileinfo; значение `Altitude` совпадает с тем, которое выводилось ранее в программе fltmc.exe.

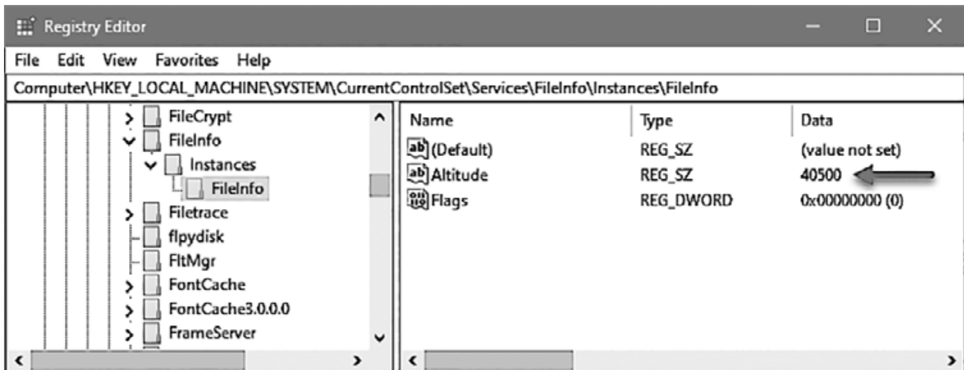


Рис. 10.4. Значение Altitude в реестре

Следующий пример показывает, почему важно значение `Altitude`. Допустим, существует мини-фильтр со значением `Altitude` 10000, задача которого — шифрование данных при записи и дешифрование при чтении. Также имеется другой мини-фильтр, задача которого — проверка данных на вредоносную активность при значении `Altitude` 9000. Схема происходящего представлена на рис. 10.5.

Драйвер шифрования шифрует входящие данные, которые должны быть записаны, после чего эти данные передаются антивирусному драйверу. У антивирусного драйвера возникают проблемы, потому что он получает зашифрованные данные, не имея реальной возможности их расшифровать (но даже если бы мог, это было бы неэффективно). В таком случае антивирусный драйвер должен иметь более высокий приоритет `Altitude`, чем у драйвера шифрования. Но как драйвер может гарантировать выполнение этого условия?

Рис. 10.5. Структура с двумя мини-фильтрами

Для этой (и других похожих) ситуации компания Microsoft определила диапазоны значений `Altitude` для драйверов, зависящие от их требований (а в конечном итоге — от их задач). Для получения правильного значения издатель драйвера должен отправить сообщение компании Microsoft (*fsfcomm@microsoft.com*) и попросить выделить драйверу значение `Altitude` в зависимости от предполагаемой цели. За полным списком диапазонов обращайтесь по ссылке[1]. По ссылке также доступен список всех драйверов, которым компания Microsoft выделила значение `Altitude`, с именем файла, значением `Altitude` и компанией-издателем.

За подробностями об оформлении запросов обращайтесь по ссылке: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/minifilter-altitude-request.

При тестировании можно выбрать любое подходящее значение Altitude без обращения в Microsoft, но для коммерческого использования следует получить официальное значение.

В табл. 10.2 приведен список групп и диапазон значений `Altitude` для каждой группы.

Таблица 10.2. Диапазоны Altitude и группы порядка загрузки

Диапазон значений Altitude	Имя группы
420 000–429 999	`Filter`
400 000–409 999	`FSFilter Top`

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes.

Диапазон значений Altitude	Имя группы
360 000–389 999	FSFilter Activity Monitor
340 000–349 999	FSFilter Undelete
320 000–329 998	FSFilter Anti-Virus
300 000–309 998	FSFilter Replication
280 000–289 998	FSFilter Continuous Backup
260 000–269 998	FSFilter Content Screener
240 000–249 999	FSFilterQuota Management
220 000–229 999	FSFilter System Recovery
200 000–209 999	FSFilter Cluster File System
180 000–189 999	FSFilter HSM
170 000–174 999	FSFilter Imaging (ex: .ZIP)
160 000–169 999	FSFilter Compression
140 000–149 999	FSFilter Encryption
130 000–139 999	FSFilter Virtualization
120 000–129 999	FSFilter PhysicalQuota management
100 000–109 999	FSFilter Open File
80 000–89 999	FSFilter Security Enhancer
60 000–69 999	FSFilter Copy Protection
40 000–49 999	FSFilter Bottom
20 000–29 999	FSFilter System

Установка

Рисунок 10.4 показывает, что существуют и другие параметры реестра, значения которых необходимо задать, помимо тех, которые можно задать стандартной функцией API `CreateService`, используемой до настоящего момента (косвенно с использованием программы sc.exe). «Правильный» способ установки минифильтра файловой системы основан на использовании INF-файлов.

INF-файлы

INF-файлы — классический механизм, который традиционно использовался для установки драйверов на базе физических устройств, но может применяться для установки драйверов любых типов. Шаблоны проекта «File System Mini-Filter», включенного в WDK, создают INF-файл, почти готовый к установке.

> Полное описание INF-файлов выходит за рамки книги. Мы ограничимся рассмотрением частей, необходимых для установки мини-фильтров файловой системы.

В INF-файлах используется старый синтаксис INI-файлов. Имеются разделы, заключенные в квадратные скобки, а в разделах следуют записи в формате «*ключ* = *значение*». Записи содержат инструкции для системы установки, которая разбирает файл. Фактически они приказывают системе установки выполнять операции двух типов: копировать файлы в определенные места и вносить изменения в реестр.

Рассмотрим INF-файл мини-фильтра файловой системы, сгенерированный мастером проекта WDK. В Visual Studio откройте раздел Device Drivers и найдите в нем подраздел Devices. Выберите вариант Filter Driver: Filesystem Mini-filter. На рис. 10.6 показано это диалоговое окно в Visual Studio 2017.

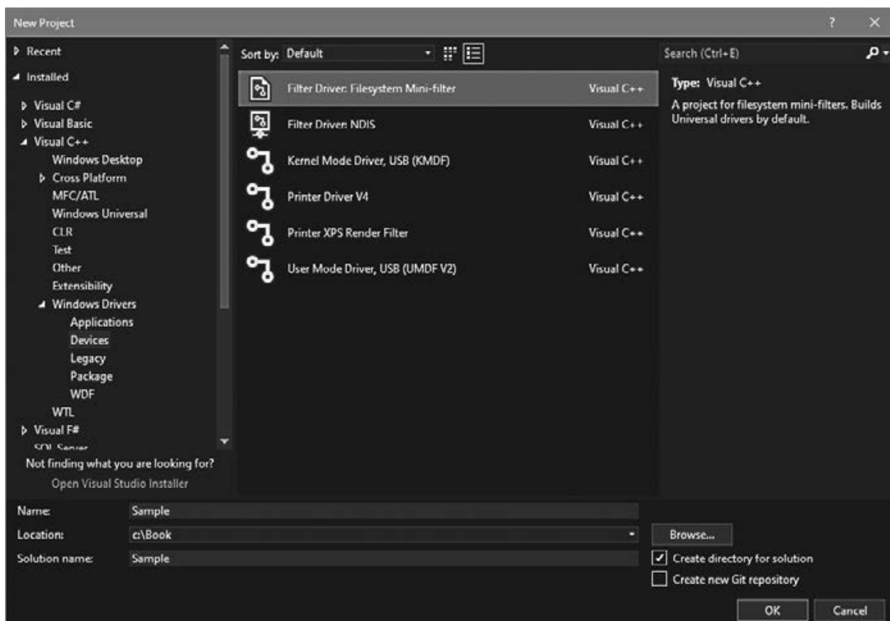


Рис. 10.6. Шаблон проекта мини-фильтра файловой системы

Введите имя проекта (Sample на рис. 10.6) и нажмите кнопку OK. В разделе Driver Files на панели Solution Explorer появляется INF-файл с именем Sample.inf.

Раздел Version

Раздел Version в INF-файлах является обязательным. Следующий файл генерируется мастером проекта WDK (слегка изменен для удобства чтения):

```
[Version]
Signature = "$Windows NT$"
; TODO - Измените Class и ClassGuid в соответствии со значением Load Order Group,
; см.https://msdn.microsoft.com/en-us/windows/hardware/gg462963
; Class = "ActivityMonitor"
;Определяется работой, которая выполняется драйвером-фильтром
; ClassGuid = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
;Это значение определяется значением Load Order Group
Class = "_TODO_Change_Class_appropriately_"
ClassGuid = {_TODO_Change_ClassGuid_appropriately_}
Provider = %ManufacturerName%
DriverVer =
CatalogFile = Sample.cat
```

Директиве Signature должна быть присвоена специальная строка «$Windows NT$». Это имя используется по историческим причинам, для нашего обсуждения они несущественны.

Директивы Class и ClassGuid являются обязательными и задают класс (тип или группу), к которому относится драйвер. Сгенерированный INF-файл содержит класс примера ActivityMonitor и связанный с ним код GUID, оба значения закомментированы (комментарии в INF-файлах следуют от символа ; до конца строки).

Простейшее решение — снять комментарии и удалить фиктивные значения Class и ClassGuid:

```
Class = "ActivityMonitor"
ClassGuid = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
```

Программа ProcessMonitor из пакета Sysinternals использует значение Altitude из группы ActivityMonitor (385 200).

Что делает директива Class? Она определяет набор заранее определенных групп устройств, используемых в основном драйверами оборудования, но ее также используют и мини-фильтры. Для мини-фильтров она приблизительно основана на группах из табл. 10.2. Полный список классов хранится в реестре в разделе HKLM\System\CurrentControlSet\Control\Class. Каждый класс однозначно идентифицируется кодом GUID; строковое имя всего лишь

упрощает идентификацию строки человеком. На рис. 10.7 показана запись класса `ActivityMonitor` в реестре.

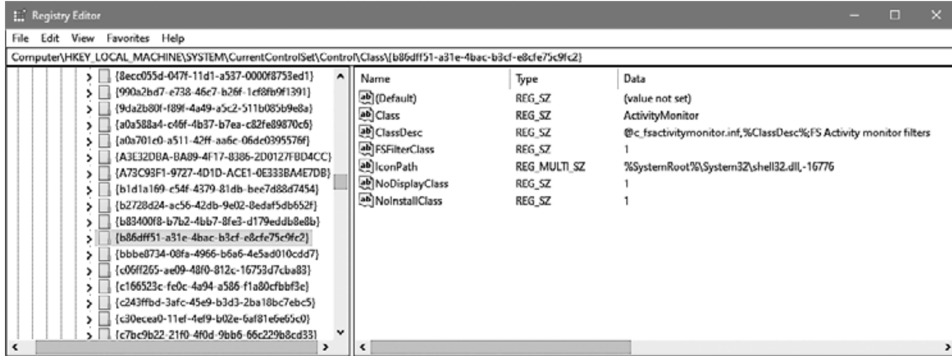


Рис. 10.7. Класс ActivityMonitor в реестре

Обратите внимание: GUID используется в качестве имени раздела. Само имя класса содержится в значении `Class`. Другие значения в разделе неважны с практической точки зрения. Параметр, который делает этот класс «актуальным» для мини-фильтров файловой системы, — `FSFilterClass` со значением 1.

Для просмотра всех существующих классов в системе можно воспользоваться программой FSClass.exe из моего Github-репозитория AllTools (https://github.com/zodiacon/AllTools). Пример выполнения этой программы:

```
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 management 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
```

> С технической точки зрения мини-фильтр файловой системы может создать собственный класс (со сгенерированным GUID), чтобы занять отдельную категорию. Тем не менее значение Altitude должно быть выбрано/отвергнуто на основании потребностей драйвера и официальных диапазонов значений Altitude (Process Monitor создает собственный класс).

Вернемся к разделу `Version` в INF-файле — директива `Provider` содержит имя издателя драйвера. Особой практической ценностью эта информация не обладает, но может отображаться в некоторых пользовательских интерфейсах, поэтому директиве лучше присвоить содержательное значение. В шаблоне WDK задается значение `%ManufacturerName%`. Все символы, заключенные между знаками `%`, являются своего рода макросами — они заменяются фактическим значением, заданным в другом разделе `Strings`. Часть этого раздела:

```
[Strings]
; TODO - Добавьте своего производителя
ManufacturerName = "Template"
ServiceDescription = "Sample Mini-Filter Driver"
ServiceName = "Sample»
```

В данном случае `ManufacturerName` будет заменяться строкой «Template». Разработчик драйвера заменит «Template» названием компании или именем продукта.

Директива `DriverVer` задает дату/время и версию драйвера. Если оставить параметры пустыми, они будут инициализированы датой сборки и номером версии, основанным на времени сборки. Директива `CatalogFile` ссылается на файл каталога, в котором хранятся цифровые подписи пакета драйвера (пакет драйвера содержит выходные файлы драйвера — обычно файлы SYS, INF и CAT).

Раздел DefaultInstall

Раздел `DefaultInstall` определяет, какие операции должны выполняться в процессе «запуска» INF-файла. С этим разделом драйвер может устанавливаться из Проводника Windows; щелкните правой кнопкой мыши на INF-файле и выберите команду Установить.

> Во внутренней реализации вызывается функция API `InstallHInfFile` с передачей пути к INF-файлу в третьем аргументе. Эта же функция может использоваться пользовательскими приложениями, которые хотят установить драйвер на программном уровне.

Раздел `DefaultInstall` не должен использоваться для установки драйверов оборудования. Он предназначен для всех остальных видов драйверов, например мини-фильтров файловой системы.

Мастер WDK сгенерировал следующий раздел:

```
[DefaultInstall]
OptionDesc = %ServiceDescription%
CopyFiles = MiniFilter.DriverFiles
```

Директива `OptionDesc` предоставляет простое описание, которое используется при установке драйвера пользователем при помощи мастера установки драйверов Plug&Play (нетипично для мини-фильтров файловой системы). Важная директива `CopyFiles` указывает на другой раздел (с указанным именем), который сообщает, какие файлы должны копироваться и в какое место.

Директива `CopyFile` указывает на раздел с именем `MiniFilter.DriverFiles`:

```
[MiniFilter.DriverFiles]
%DriverName%.sys
```

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

Раздел `DestinationDirs` предоставляет следующую информацию:

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

`DefaultDestDir` — каталог, используемый по умолчанию для копирования, если он не задан явно. Заданное значение выглядит немного странно (12), но на самом деле это специальное число, обозначающее системный каталог драйверов, указанный в комментарии второй директивы. Каталог System32\

Drivers — каноническое место для размещения драйверов. В предыдущих главах мы размещали драйверы практически где угодно, но драйверы следует размещать в папке drivers хотя бы для защиты, так как эта папка является системной и ограничивает доступ для стандартных пользователей.

Другие «волшебные» числа, обозначающие специальные каталоги, перечислены ниже. Полный список доступен по ссылке[1].

- 10 — каталог Windows (%SystemRoot%).
- 11 — каталог System (%SystemRoot%\System32).
- 24 — корневой каталог системного диска (например, C:\).
- 01 — каталог, из которого был прочитан INF-файл.

Если эти числа используются в составе пути, они должны быть заключены между символами % (например, %10%\Config).

Вторая директива (`MiniFilter.DriverFiles`) указывает, что выходные файлы должны быть скопированы в целевой каталог, обозначенный тем же специальным числом.

> Вероятно, вы не сразу привыкнете к структуре и семантике INF-файлов. Это не плоский файл, а иерархический. Одна директива может ссылаться на другой раздел, в котором другая директива ссылается на другой раздел и т. д. Я бы предпочел, чтобы компания Microsoft отказалась от старого формата на базе INI-файлов и перешла на формат, иерархический по своей природе (например, XML или JSON).

Раздел Service

Следующие разделы, которые представляют для нас интерес, относятся к записи в раздел реестра `services` (сходный с тем, что делает `CreateService`). Эта операция обязательна для каждого драйвера. Все начинается со следующего определения:

```
[DefaultInstall.Services]
AddService = %ServiceName%,,MiniFilter.Service
```

Раздел `DefaultInstall.Services` ищется автоматически. Если он будет найден, то директива `AddService` указывает на другой раздел с описанием информации, записываемой в раздел реестра с именем `%ServiceName%`. Дополнительная

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/using-dirids.

запятая резервирует место для набора флагов; в данном случае значение равно нулю. Сгенерированный мастером код выглядит так:

```
[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 - Изменить значение Load Order Group
; LoadOrderGroup = "FSFilter Activity Monitor"
LoadOrderGroup = "_TODO_Change_LoadOrderGroup_appropriately_"
AddReg           = MiniFilter.AddRegistry
```

Вероятно, вы узнаете многие директивы, соответствующие значениям в разделе реестра `services`. Значение `ServiceType` равно 2 для драйверов, связанных с файловой системой (в отличие от 1 для «стандартных» драйверов). Зависимости (`Dependencies`) ранее нам еще не встречались; это список служб/драйверов, от которых зависит текущая служба/драйвер. В случае мини-фильтра файловой системы это сам диспетчер фильтров.

Значение `LoadOrderGroup` должно задаваться на основании имен групп мини-фильтров из табл. 10.2. Наконец, директива `AddReg` указывает на другой раздел с инструкциями по добавлению новых записей реестра. Доработанный раздел `MiniFilter.Service` выглядит так:

```
[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 сколько угодно) используется для добавления нестандартных записей реестра для произвольных целей. INF-файл, сгенерированный мастером, содержит следующие добавления в реестр:

```
[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 каждой записи содержит следующие составляющие (в указанном порядке):

- Корневой раздел — одно из значений HKLM, HKCU (текущий пользователь), HKCR (классы), HKU (пользователи) или HKR (относительно вызывающего раздела). В данном случае HKR — подраздел службы (HKLMSystemCurrentControlSetServicesSample).
- Подраздел корневого раздела (если не задан, используется сам раздел).
- Имя присваиваемого значения.
- Флаги — определено достаточно много, по умолчанию используется значение 0 (признак записи параметра REG_SZ). Примеры других флагов:
  - 0x100000 — запись REG_MULTI_SZ;
  - 0x100001 — запись REG_DWORD;
  - 0x000001 — запись двоичного значения (REG_BINARY);
  - 0x000002 — запрет на перезапись существующего значения;
  - 0x000008 — присоединение значения. Текущим должно быть значение REG_MULTI_SZ;
  - Фактическое значение или значения для записи/присоединения.

Приведенный выше фрагмент назначает некоторые значения по умолчанию для мини-фильтров файловой системы. Самое важное значение — приоритет Altitude — берется из параметра %Instance1.Altitude% в разделе Strings.

Завершение INF-файла

Остается внести последнее изменение — значение Altitude в разделе Strings. В нашем примере раздел выглядит так:

```
[Strings]
; другие записи
; Информация, относящаяся к конкретному экземпляру
DefaultInstance         = "Sample Instance"
Instance1.Name          = "Sample Instance"
Instance1.Altitude      = "360100"
Instance1.Flags         = 0x0 ; Разрешить все присоединения
```

Значение Altitude было выбрано из диапазона Altitude для группы Activity Monitor. В реальном драйвере вы получите это число от компании Microsoft по запросу, как упоминалось ранее.

Наконец, флаги указывают, что драйвер может присоединяться к любому тому, но в реальности драйвер будет получать запросы через функцию обратного вызова, в которой он сможет разрешить или отклонить присоединение.

Установка драйвера

После того как в INF-файл будут внесены изменения, а код драйвера будет откомпилирован, все готово к установке. Простейший способ установки — скопировать пакет драйвера (файлы SYS, INF и CAT) в целевую систему, щелкнуть правой кнопкой мыши на INF-файле в «Проводнике» и выбрать команду Install. Команда «запустит» INF-файл и выполнит все необходимые операции.

На этой стадии мини-фильтр установлен, и его можно загрузить программой командной строки fltmc (предполагается, что драйверу присвоено имя `sample`):

```
c:\>fltmc load sample
```

Обработка операций ввода/вывода

Основная функция мини-фильтра файловой системы — обработка операций ввода/вывода посредством реализации обратных вызовов перед и/или после операции, представляющей интерес. Обратные вызовы перед операцией позволяют мини-фильтру полностью отклонить операцию, тогда как обратные вызовы после операции позволяют просмотреть результат операции и в некоторых случаях — внести изменения в возвращаемую информацию.

Обратные вызовы перед операцией

Все обратные вызовы перед операцией имеют одинаковый прототип:

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

Рассмотрим возможные возвращаемые значения от обратного вызова перед операцией, представленные перечислением `FLT_PREOP_CALLBACK_STATUS`. Некоторые возвращаемые значения, часто используемые на практике:

- `FLT_PREOP_COMPLETE` — означает, что драйвер завершает операцию. Диспетчер фильтров не активизирует функцию обратного вызова после операции (если она зарегистрирована) и не передает запрос мини-фильтрам нижних уровней.
- `FLT_PREOP_SUCCESS_NO_CALLBACK` — означает, что функция обратного вызова перед операцией завершила обработку запроса и позволяет ему перейти к следующему фильтру. Драйвер не хочет, чтобы для этой операции была активизирована функция обратного вызова после операции.

- FLT_PREOP_SUCCESS_WITH_CALLBACK — означает, что драйвер позволяет диспетчеру фильтров передать запрос фильтрам нижних уровней, но он хочет, чтобы для этой операции была активизирована функция обратного вызова после операции.
- FLT_PREOP_PENDING — означает, что драйвер приостанавливает операцию. Диспетчер фильтров не продолжает обработку запроса до того момента, когда драйвер вызовет FltCompletePendedPreOperation; этот вызов сообщает диспетчеру фильтров, что обработку запроса можно продолжить.
- FLT_PREOP_SYNCHRONIZE — аналог FLT_PREOP_SUCCESS_WITH_CALLBACK, но драйвер приказывает диспетчеру фильтров активизировать свою функцию обратного вызова после операции в том же потоке на уровне IRQL <= APC_LEVEL (обычно функция обратного вызова после операции может активизироваться на уровне IRQL <= DISPATCH_LEVEL в произвольном потоке).

Аргумент Data предоставляет всю информацию, связанную с самой операцией ввода/вывода, в виде структуры FLT_CALLBACK_DATA, которая определяется следующим образом:

```
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 обратном вызове после операции. Краткая сводка важнейших полей структуры:

- Flags — ноль или комбинация следующих значений:
    - FLTFL_CALLBACK_DATA_DIRTY — означает, что драйвер внес изменения в структуру, а затем вызвал FltSetCallbackDataDirty. Изменить можно каждое поле структуры, кроме Thread и RequestorMode.
    - FLTFL_CALLBACK_DATA_FAST_IO_OPERATION — означает, что операция является операцией быстрого ввода/вывода.

- FLTFL_CALLBACK_DATA_IRP_OPERATION — означает, что операция является операцией на базе IRP.
- FLTFL_CALLBACK_DATA_GENERATED_IO — означает, что операция сгенерирована другим мини-фильтром.
- FLTFL_CALLBACK_DATA_POST_OPERATION — означает, что обратный вызов активизируется после операции (а не перед операцией).

- Thread — непрозрачный указатель на поток, который запросил выполнение операции.
- IoStatus — статус запроса. Обратный вызов перед операцией может задать это значение, а затем указать, что обработка перед операцией завершена, вернув FLT_PREOP_COMPLETE. Обратный вызов после операции может проверить итоговый статус операции.
- RequestorMode — сообщает, поступил ли запрос из пользовательского режима (UserMode) или режима ядра (KernelMode).
- Iopb — структура, содержащая подробные параметры запроса. Определение структуры выглядит так:

```
    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;
```

Самые полезные поля этой структуры:

- TargetFileObject — объект файла, который является целью операции; значение может быть полезно при вызове некоторых функций API.
- Parameters — огромное объединение с данными конкретной операции (на концептуальном уровне напоминает поле Parameters структуры IO_STACK_LOCATION). Чтобы получить необходимую информацию, драйвер проверяет соответствующую структуру в этом объединении. Некоторые из этих структур будут рассмотрены позднее в этой главе, когда мы займемся конкретными типами операций.

Во втором аргументе обратного вызова перед операцией передается другая структура типа FLT_RELATED_OBJECTS. Эта структура в основном содержит непрозрачные дескрипторы текущего фильтра, экземпляра и тома, которые могут использоваться некоторыми функциями API. Полное определение структуры выглядит так:

```
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` содержит тот же объект, который хранится в поле `TargetFileObject` блока параметров ввода/вывода.

В последнем аргументе обратного вызова перед операцией передается контекст, который может задаваться драйвером. Если значение задано, оно передается функции обратного вызова после операции для того же запроса (по умолчанию используется значение `NULL`).

Обратные вызовы после операции

Все обратные вызовы после операции имеют одинаковый прототип:

```
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);
```

Функция после операции вызывается на уровне IRQL <= `DISPATCH_LEVEL` в контексте произвольного потока, если только обратный вызов перед операцией не вернет `FLT_PREOP_SYNCHRONIZE`; в этом случае диспетчер фильтров гарантирует, что обратный вызов после операции будет вызван на уровне IRQL < `DISPATCH_LEVEL` в том же потоке, в котором был выполнен обратный вызов перед операцией.

В первом случае драйвер не может выполнить некоторые виды операций из-за слишком высокого уровня IRQL:

- Драйвер не может обращаться к выгружаемой памяти.
- Драйвер не может использовать функции API режима ядра, работающие только на уровне IRQL < `DISPATCH_LEVEL`.
- Драйвер не может захватывать примитивы синхронизации: мьютексы, быстрые мьютексы, ресурсы исполнительной системы, семафоры, события и т. д. (при этом он может захватывать спин-блокировки).
- Драйвер не может устанавливать, получать или удалять контексты (см. раздел «Контексты» этой главы), но может освобождать контексты.

Если драйверу понадобится сделать что-то из вышеперечисленного, он должен каким-то образом передать выполнение в другую функцию, вызываемую на уровне IRQL < DISPATCH_LEVEL. Это можно сделать одним из двух способов:

- Драйвер вызывает функцию FltDoCompletionProcessingWhenSafe, настраивающую функцию обратного вызова, которая вызывается системным рабочим потоком на уровне IRQL < DISPATCH_LEVEL (если обратный вызов после операции был активизирован на уровне IRQL = DISPATCH_LEVEL).
- Драйвер отправляет рабочий элемент вызовом FltQueueDeferredIoWorkItem. Функция ставит в очередь рабочий элемент, который со временем будет выполнен системным рабочим потоком на уровне IRQL = PASSIVE_LEVEL. В обратном вызове рабочего элемента драйвер вызовет FltCompletePendedPostOperation, которая сообщит диспетчеру фильтров о завершении активности после операции.

Хотя вариант с FltDoCompletionProcessingWhenSafe проще, у него есть некоторые ограничения, не позволяющие использовать его в некоторых ситуациях:

- Он не может использоваться для IRP_MJ_READ, IRP_MJ_WRITE или IRP_MJ_FLUSH_BUFFERS, потому что завершение этих операций синхронно с нижним уровнем может привести к взаимной блокировке.
- Он может вызываться только для операций на базе IRP (для проверки можно воспользоваться макросом FLT_IS_IRP_OPERATION).

В любом случае использование механизмов отложенного выполнения невозможно, если в аргументе флагов включен режим FLTFL_POST_OPERATION_DRAINING, означающий, что обратный вызов после операции является частью процедуры отсоединения тома. В данном случае обратный вызов после операции активизируется на уровне IRQL < DISPATCH_LEVEL.

Хотя на первый взгляд кажется, что для выполнения обратного вызова после операции в удобном контексте достаточно вернуть FLT_PREOP_SYNCHRONIZE из обратного вызова перед операцией, такое решение сопряжено с некоторыми лишними затратами ресурсов, которых драйверы должны избегать насколько это возможно.

Обратный вызов после операции создания (IRP_MJ_CREATE) гарантированно будет активизирован запрашивающим потоком на уровне IRQL PASSIVE_LEVEL.

Обратный вызов после операции обычно возвращает значение FLT_POSTOP_FINISHED_PROCESSING, которое сообщает, что драйвер завершил обработку операции. Но если драйверу необходимо выполнить некоторую работу в рабочем

элементе (например, из-за высокого уровня IRQL), он может вернуть значение `FLT_POSTOP_MORE_PROCESSING_REQUIRED`. Это значение сообщает диспетчеру фильтров, что операция еще не завершена, а в рабочем элементе вызывается функция `FltCompletePendedPostOperation`, которая сообщает диспетчеру фильтров о том, что он может продолжить обработку этого запроса.

> В ходе обработки приходится учитывать множество технических подробностей, а в документации WDK их содержится еще больше. Некоторые из описанных механизмов будут использованы позднее в этой главе.

Драйвер Delete Protector

Пришло время воплотить часть новой информации, приведенной в этой главе, в реальный драйвер. Тот драйвер, который мы создадим, сможет защищать некоторые файлы от удаления определенными процессами. Драйвер будет построен на основе шаблона проекта из WDK (несмотря на то что мне не нравится часть кода, сгенерированного этим шаблоном).

Для начала создайте новый проект мини-фильтра файловой системы с именем `DelProtect` (или другим именем на ваш выбор) и поручите мастеру сгенерировать исходные файлы и код.

Затем нужно заняться INF-файлом. Драйвер принадлежит к классу «Undelete» (что выглядит разумно с учетом его предназначения), поэтому мы выберем значение `Altitude` из этого диапазона. Измененные части INF-файла:

```
[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"

; Информация, относящаяся к конкретному экземпляру.
DefaultInstance         = "DelProtect Instance"
Instance1.Name          = "DelProtect Instance"
Instance1.Altitude      = "345101" ; из диапазона группы undelete
Instance1.Flags         = 0x0      ; Разрешить все присоединения
```

Разобравшись с INF-файлом, можно обратиться к коду. Сгенерированный файл с исходным кодом называется DelProtect.c; переименуйте его в DelProtect.cpp, чтобы мы могли свободно использовать C++.

Функция `DriverEntry`, предоставляемая шаблоном проекта, уже содержит код регистрации мини-фильтра. Обратные вызовы необходимо слегка подправить и указать, какие именно из них нас интересуют. Таким образом, необходимо найти ответ на вопрос: какие первичные функции задействованы в удалении файлов?

Существуют два способа удаления файлов. Первый основан на использовании операции `IRP_MJ_SET_INFORMATION`. Эта операция предоставляет целый набор операций, и удаление — всего лишь одна из них. Второй (пожалуй, наиболее распространенный) способ удаления файлов основан на открытии файла с флагом `FILE_DELETE_ON_CLOSE`. Файл будет удален при закрытии последнего дескриптора для него.

> Флаг может быть установлен в пользовательском режиме для `CreateFile` в составе набора флагов (предпоследний аргумент). Этот же флаг используется во внутренней реализации функции более высокого уровня `DeleteFile`.

Чтобы драйвер правильно работал в любых ситуациях, необходимо поддерживать оба варианта. Массив `FLT_OPERATION_REGISTRATION` следует изменить для поддержки обоих вариантов:

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

Конечно, необходимо реализовать функции `DelProtectPreCreate` и `DelProtectPreSetInformation` соответствующим образом. Обе функции являются обратными вызовами перед операцией создания, так как мы хотим отклонять эти запросы при некоторых условиях.

Обратные вызовы перед созданием

Начнем с создания функции перед созданием драйвера (она устроена чуть проще). Ее прототип выглядит так же, как у любого другого обратного вызова перед операцией (просто скопируйте его из обобщенной функции `DelProtectPreOperation`, предоставленной шаблоном проекта):

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

Сначала проверим, исходит ли операция из режима ядра, и если проверка даст положительный результат, просто позволим операции беспрепятственно продолжаться:

```
UNREFERENCED_PARAMETER(CompletionContext);
UNREFERENCED_PARAMETER(FltObjects);

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

Конечно, это не обязательно, но в большинстве случаев лучше не мешать коду ядра делать то, что может быть очень важным.

Затем необходимо проверить, присутствует ли в запросе на создание флаг `FILE_DELETE_ON_CLOSE`. Структура, которая нас интересует, — поле `Create` в структуре `Parameters` внутри `Iopb`:

```
const auto& params = Data->Iopb->Parameters.Create;
if (params.Options & FILE_DELETE_ON_CLOSE) {
    // Операция удаления
}
// Иначе просто продолжить
return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Переменная `params` ссылается на структуру `Create`, которая определяется следующим образом:

```
struct {
    PIO_SECURITY_CONTEXT SecurityContext;
    //
    // Младшие 24 разряда содержат значения флага CreateOptions.
    // Старшие 8 разрядов содержат значения CreateDisposition.
    //
    ULONG Options;

    USHORT POINTER_ALIGNMENT FileAttributes;
    USHORT ShareAccess;
    ULONG POINTER_ALIGNMENT EaLength;
```

```
    PVOID EaBuffer;              //Не входит в список параметров IO_STACK_
                                 //LOCATION
    LARGE_INTEGER AllocationSize; //Не входит в список параметров IO_STACK_
                                 //LOCATION
} Create;
```

В общем случае для любой операции ввода/вывода следует обращаться к документации, чтобы понять, какие возможности вам доступны и как ими пользоваться. В нашем случае поле `Options` содержит комбинацию флагов, документированную в описании функции `FltCreateFile` (которая будет использоваться позднее в этой главе в другом контексте).

Код проверяет, существует ли этот флаг, и если существует, значит, инициирована операция удаления. Наш драйвер будет блокировать операции удаления, исходящие от процессов cmd.exe. Для этого необходимо получить путь к образу вызывающего процесса. Так как операция создания активизируется синхронно, мы знаем, что вызывающей стороной является процесс, пытающийся что-то удалить. Но как получить путь к образу текущего процесса?

Одно из возможных решений в этой ситуации — использовать функцию API `NtQueryInformationProcess` режима ядра (или ее `Zw`-эквивалент — `ZwQueryInformationProcess`). Функция является отчасти документированной, а ее прототип доступен в заголовке пользовательского режима `<wintrnl.h>`. Мы можем просто скопировать ее объявление и преобразовать его в `Zw` в исходном коде:

```
extern "C" NTSTATUS ZwQueryInformationProcess(
    _In_      HANDLE           ProcessHandle,
    _In_      PROCESSINFOCLASS ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength);
```

Перечисление `PROCESSINFOCLASS` большей частью доступно в `<ntddk.h>`. Я говорю «большей частью», потому что этом файле указаны не все поддерживаемые значения. (Мы вернемся к этой проблеме в главе 11.)

Для целей нашего драйвера можно использовать значение `ProcessImageFileName` из перечисления `PROCESSINFOCLASS`, которое позволяет получить полный путь к файлу образа процесса. Этот путь можно будет сравнить с путем к cmd.exe.

В документации к функции `NtQueryInformationProcess` сказано, что для `ProcessImageFileName` возвращается структура `UNICODE_STRING`, память для которой должна быть выделена вызывающей стороной:

```
auto size = 300; // Произвольный размер, достаточный для хранения пути к образу
                    cmd.exe
auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
if (processName == nullptr)
```

```
    return FLT_PREOP_SUCCESS_NO_CALLBACK;
RtlZeroMemory(processName, size); // Строка должна завершаться NULL-символом
```

Обратите внимание: мы не ограничиваемся выделением памяти по размеру структуры `UNICODE_STRING`. Тогда где функция API должна хранить саму строку? Мы выделяем непрерывный буфер, а функция API разместит символы после самой структуры в памяти. Теперь можно вызвать функцию API:

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

Макрос `NtCurrentProcess` возвращает псевдодескриптор, ссылающийся на текущий процесс (по аналогии с функцией API пользовательского режима `GetCurrentProcess`).

> Под «псевдодескриптором» понимается дескриптор, который не нужно (и невозможно) закрывать.

Если вызов завершается успехом, необходимо сравнить имя файла образа процесса с cmd.exe. Одно из простых решений:

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

Сравнение получается не настолько простым, насколько нам хотелось бы. Фактический путь, возвращаемый вызовом `ZwQueryInformationProcess`, имеет вид \Device\HarddiskVolume3\Windows\System32\cmd.exe. В своем простом драйвере мы ограничиваемся поиском подстроки «System32\cmd.exe» или «SysWOW64\cmd.exe» (последнее — на случай вызова 32-разрядной версии cmd.exe). Кстати говоря, при сравнении учитывается регистр символов. Такой способ сравнения не идеален: что, если регистр символов будет нарушен? Что, если кто-то скопирует файл cmd.exe в другую папку и запустит его оттуда? На самом деле это должен решать драйвер. С таким же успехом можно включить в сравнение только cmd.exe. Для нашего простейшего драйвера этого достаточно.

Если это действительно файл cmd.exe, необходимо предотвратить успешное выполнение операции. Обычно это делается заменой статуса операции (`Data->IoStatus.Status`) соответствующим статусом ошибки и возвращением его из обратного вызова `FLT_PREOP_COMPLETE`, чтобы сообщить диспетчеру фильтров, что продолжать обработку запроса не следует.

Полный код обратного вызова перед созданием с небольшими изменениями:

```
_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) {
        // Операция удаления
        KdPrint(("Delete on close: %wZ\n", &Data->Iopb->
TargetFileObject->FileName));

        auto size = 300; // Произвольный размер, достаточный для хранения пути
                         // к образу cmd.exe
        auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
        if (processName == nullptr)
            return FLT_PREOP_SUCCESS_NO_CALLBACK;

        RtlZeroMemory(processName, size); // Строка должна завершаться
                                          // NULL-символом
        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) {
                // Отказ в выполнении запроса
                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;
}
```

SAL-аннотация `_Use_decl_annotations_` означает, что настоящие SAL-аннотации находятся в объявлении функции, а не в объявлении и реализации. Она просто немного упрощает чтение реализации.

Чтобы построить драйвер, необходимо предоставить реализацию обратного вызова перед операцией `IRP_MJ_SET_INFORMATION`. Пример простой реализации, которая разрешает 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;
}
```

Драйвер можно построить и развернуть, а затем протестировать следующей командой:

```
del somefile.txt
```

Вы увидите, что хотя управление передается обработчику `IRP_MJ_CREATE` и драйвер отказывает в выполнении операции, файл все равно успешно удаляется. Дело в том, что файл cmd.exe действует хитро: если одна попытка завершается неудачей, он пробует другой способ. После отказа попытки `FILE_DELETE_ON_CLOSE` используется способ с `IRP_MJ_SET_INFORMATION`, а поскольку все операции с `IRP_MJ_SET_INFORMATION` разрешены, операция завершается успешно.

Обработка информации перед операцией

Все готово к реализации обратного вызова для `IRP_MJ_SET_INFORMATION`, чтобы учесть все варианты и обработать второй способ удаления файлов в файловых системах. Для начала все вызовы из режима ядра будут игнорироваться, как и в случае 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;
```

Так как запрос `IRP_MJ_SET_INFORMATION` может использоваться для выполнения нескольких типов операций, необходимо проверить, действительно ли операция является операцией удаления. Драйвер должен сначала обратиться к правильной структуре в объединении параметров, которая объявляется следующим образом:

```
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;
```

Поле `FileInformationClass` указывает, какой тип операции представляет данный экземпляр, поэтому необходимо проверить, является ли операция операцией удаления:

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

if (params.FileInformationClass != FileDispositionInformation &&
    params.FileInformationClass != FileDispositionInformationEx) {
    // Не является операцией удаления
    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

Значение перечисления `FileDispositionInformation` обозначает операцию удаления. Аналогичная структура `FileDispositionInformationEx` не документирована, но она используется во внутренней реализации функцией пользовательского режима `DeleteFile`, поэтому проверяем оба варианта.

Если операция является операцией удаления, необходимо выполнить еще одну проверку: обратиться к информационному буферу типа `FILE_DISPOSITION_INFORMATION` для операций удаления и проверить хранящееся в ней логическое значение:

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

Проверка закончена: выполняется операция удаления. В случае `IRP_MJ_CREATE` обратный вызов активизируется потоком — источником запроса (а следовательно, процессом-источником), поэтому мы можем просто обратиться к текущему процессу, чтобы узнать, какой файл образа использовался для вызова. Для всех остальных первичных функций выполнение этого условия не гарантировано, и мы должны проверить поле `Thread` в предоставленных данных для стороны вызова. По этому потоку можно получить указатель на процесс:

```
// От какого процесса получен запрос?
auto process = PsGetThreadProcess(Data->Thread);
NT_ASSERT(process); // На самом деле всегда проходит успешно
```

Наша цель — вызов ZwQueryInformationProcess, но для этого необходимо получить дескриптор. Здесь в игру вступает функция ObOpenObjectByPointer, которая позволяет получить дескриптор для объекта. Она определяется следующим образом:

```
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 перечислены ниже:

- Object — объект, для которого нужно получить дескриптор. Это может быть объект режима ядра любого типа.
- HandleAttributes — набор необязательных флагов. Самый полезный флаг — OBJ_KERNEL_HANDLE (другие флаги будут описаны в главе 11). С этим флагом возвращаемый дескриптор является дескриптором режима ядра, который не может использоваться в коде пользовательского режима и может использоваться из контекста любого процесса.
- PassedAccessState — необязательный указатель на структуру ACCESS_STATE, которая обычно не используется в драйверах (присваивается NULL).
- DesiredAccess — маска доступа, с которой должен быть открыт дескриптор. Если аргумент AccessMode содержит KernelMode, это может быть ноль, а возвращаемый дескриптор обладает всеми полномочиями.
- ObjectType — необязательный тип объекта, с которым функция должна сравнить Object — например, *PsProcessType, *PsThreadType и другие экспортированные типы объектов. Со значением NULL никакие проверки с переданным объектом не выполняются.
- AccessMode — может содержать UserMode или KernelMode. Драйверы обычно используют KernelMode, чтобы указать, что запрос не делается от имени процесса пользовательского режима. Со значением KernelMode никакие проверки прав доступа не выполняются.
- Handle — указатель на возвращенный дескриптор.

Для вышеприведенной функции дескриптор для процесса может быть открыт 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;
```

После получения дескриптора процесса можно запросить имя файла образа процесса и сравнить его с cmd.exe:

```
    auto returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
    auto size = 300;
    auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
    if (processName) {
        RtlZeroMemory(processName, size); // Строка должна завершаться
                                          // NULL-символом
        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 можно протестировать законченный драйвер. Вы увидите, что cmd.exe не может удалять файлы — при попытке удаления возвращается ошибка «Отказано в доступе».

Небольшой рефакторинг

Два реализованных нами обратных вызова перед операцией содержат много общего кода, поэтому в соответствии с принципом DRY («Don’t Repeat Yourself», то есть «не повторяйтесь») можно выделить код открытия дескриптора процесса, получения файла образа и сравнения с cmd.exe в отдельную функцию:

```
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 получает непрозрачный указатель на процесс, пытающийся удалить файл. Если адрес процесса соответствует текущему процессу (`PsGetCurrentProcess`), то полноценное открытие — простая трата времени, и вместо этого проще воспользоваться псевдодескриптором `NtCurrentProcess`. В противном случае требуется полное открытие процесса. Будьте внимательны: чтобы избежать утечки ресурсов, не забудьте освободить буфер файла образа и закрыть дескриптор процесса (если он был открыт).

Теперь можно подключить вызов этой функции к обработке `IRP_MJ_CREATE`. Переработанная функция выглядит так:

```
_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) {
        // Операция удаления
        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 переработанная версия обратного вызова перед операцией для `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) {
        // Не является операцией удаления
        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;

    // От какого процесса поступил запрос?
    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;
}
```

Построение обобщенной версии драйвера

Текущий драйвер проверяет только операции удаления от cmd.exe. Обобщим драйвер, чтобы в нем можно было регистрировать имена исполняемых файлов, операции удаления из которых можно было бы запрещать.

Для этого мы создадим «классический» объект устройства и символическую ссылку под аналогии с тем, как это делалось в предыдущих главах. Сделать это несложно, а драйвер может как решать задачи мини-фильтра файловой системы, так и предоставлять объект CDO (Control Device Object).

Для простоты имена файлов будут храниться в массиве фиксированного размера, который будет защищаться быстрым мьютексом, как в предыдущих главах. Также будут задействованы созданные ранее обертки `FastMutex` и `AutoLock`. Новые глобальные переменные:

```
const int MaxExecutables = 32;

WCHAR* ExeNames[MaxExecutables];
int ExeNamesCount;
FastMutex ExeNamesLock;
```

Функциональность новой версии `DriverEntry` расширилась — теперь она создает объект устройства и символическую ссылку, задает функции диспетчеризации и регистрируется 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;
```

Определим несколько кодов управляющих операций 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 этих кодов управляющих операций нет ничего нового — ниже приведен полный код функции диспетчеризации `IRP_MJ_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 коде не хватает вспомогательных функций `FindExecutable` и `ClearAll`, которые определяются следующим образом:

```
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;
}
```

С этим кодом необходимо внести изменения: в обратный вызов перед операцией создания поиск имен исполняемых файлов в массиве. Обновленная версия кода выглядит так:

```
_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) {
        // Операция удаления
        KdPrint(("Delete on close: %wZ\n", &FltObjects->FileObject->FileName));

        auto size = 512; // произвольный размер
        auto processName = (UNICODE_STRING*)ExAllocatePool(PagedPool, size);
```

```
        if (processName == nullptr)
            return FLT_PREOP_SUCCESS_NO_CALLBACK;

        RtlZeroMemory(processName, size);
        auto status = ZwQueryInformationProcess(NtCurrentProcess(), \
ProcessImageFileName,
            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)) { // Пропустить \
                Data->IoStatus.Status = STATUS_ACCESS_DENIED;
                KdPrint(("Prevented delete in IRP_MJ_CREATE\n"));
                returnStatus = FLT_PREOP_COMPLETE;
            }
        }
        ExFreePool(processName);
    }
    return returnStatus;
}
```

Главное изменение в этом коде — вызов `FindExecutable` для определения того, входит ли имя исполняемого образа текущего процесса в число значений, хранящихся в массиве. Если оно будет найдено в массиве, необходимо установить статус «отказано в доступе» и вернуть код `FLT_PREOP_COMPLETE`.

Тестирование измененного драйвера

Ранее мы тестировали драйвер, удаляя файлы из cmd.exe, но этот способ может быть недостаточно общим, поэтому лучше создать собственное тестовое приложение. Есть три способа удаления файлов функциями API пользовательского режима:

1. Вызов функции `DeleteFile`.
2. Вызов функции `CreateFile` с флагом `FILE_FLAG_DELETE_ON_CLOSE`.
3. Вызов функции `SetFileInformationByHandle` для открытого файла.

Во внутренней реализации существуют только два способа удаления файлов — `IRP_MJ_CREATE` с флагом `FILE_DELETE_ON_CLOSE` и `IRP_MJ_SET_INFORMATION` со структурой `FileDispositionInformation`. Очевидно, в приведенном списке пункт (2) соответствует первому варианту, а пункт (3) — второму. Вопросы остаются только с `DeleteFile` — как эта функция удаляет файл?

С точки зрения драйвера это совершенно неважно, так как в любом случае этот способ должен соответствовать одному из двух вариантов, поддерживаемых драйвером. Для любознательных читателей скажу, что `DeleteFile` использует `IRP_MJ_SET_INFORMATION`.

Мы создадим проект консольного приложения DelTest, которое должно использоваться примерно так:

```
c:\book>deltest
Usage: deltest.exe <method> <filename>
    Method: 1=DeleteFile, 2=delete on close, 3=SetFileInformation.
```

Рассмотрим код пользовательского режима для каждого из этих способов (предполагается, `filename` — переменная, указывающая на имя файла, переданное в командной строке).

Вариант с `DeleteFile` реализуется тривиально:

```
BOOL success = ::DeleteFile(filename);
```

Открытие файла с флагом удаления при закрытии выполняется так:

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

При закрытии дескриптора файл должен быть удален (если драйвер не предотвратит это!).

Остается вызвать функцию `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);
```

При наличии такого инструмента мы сможем протестировать свой драйвер. Несколько примеров:

```
C:\book>fltmc load delprotect2

C:\book>DelProtectConfig.exe add deltest.ex
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 необходимо знать имя файла, к которому обращен запрос. На первый взгляд узнать эту информацию несложно: структура `FILE_OBJECT` содержит поле `FileName`, в котором она должна содержаться.

К сожалению, все не так просто. Файлы могут открываться как с полным, так и с относительным путем; несколько операций переименования могут выполняться с одним файлом одновременно; часть информации об именах файлов кэшируется. По этим и другим внутренним причинам содержимому поля `FileName` в объекте файла доверять не следует. Оно заведомо действительно только в обратном вызове перед операцией `IRP_MJ_CREATE`, и даже в этом случае не обязательно в том формате, который нужен драйверу.

Для компенсации диспетчер фильтров предоставляет функцию API `FltGetFileNameInformation`, которая может вернуть правильное имя файла в случае необходимости. Прототип этой функции:

```
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` содержит набор флагов, которые определяют (среди прочего) нужный формат файла. В большинстве драйверов используется значение `FLT_FILE_NAME_NORMALIZED` (полный путь), объединенное операцией OR с `FLT_FILE_NAME_QUERY_DEFAULT` (поиск имени в кэше, если не найдено — запрос

к файловой системе). Результат вызова предоставляется последним параметром `FileNameInformation`. Это структура, для которой выделяется память и которая должна быть освобождена вызовом `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;
```

Основные компоненты — несколько структур `UNICODE_STRING` для хранения различных составляющих имени файла. Изначально только поле `Name` инициализируется полным именем файла (в зависимости от флагов, используемых при запросе информации имени файла, «полное» имя может оказаться частичным). Если в запросе был задан флаг `FLT_FILE_NAME_NORMALIZED`, то `Name` указывает на полное имя в форме имени устройства. Термин «имя устройства» означает, что файл вида c:\mydir\myfile.txt хранится под внутренним именем устройства, которому соответствует «C:» например \Device\HarddiskVolume3\mydir\myfile.txt. Это усложняет задачу драйвера, если он каким-то образом зависит от путей, предоставляемых пользовательским режимом (подробнее об этом позднее).

Драйвер никогда не должен изменять эту структуру, потому что диспетчер фильтров кэширует ее для использования другими драйверами.

Так как по умолчанию предоставляется только полное имя (поле `Name`), часто бывает необходимо разбивать полное имя на составляющие. К счастью, диспетчер фильтров предоставляет такую возможность в виде функции API `FltParseFileNameInformation`. Эта функция получает объект `FLT_FILE_NAME_INFORMATION` и заполняет другие поля `UNICODE_STRING` в структуре.

> Обратите внимание: функция `FltParseFileNameInformation` никакой памяти не выделяет. Она просто присваивает полям `Buffer` и `Length` каждой структуры `UNICODE_STRING` ссылки на соответствующую часть полного имени `Name`. Это означает, что никакой функции «отмены разбора» имени не существует, и она не нужна.

В тех сценариях, в которых полный путь представлен простой строкой C, более простая (и слабая) функция `FltParseFileName` позволит легко получить доступ к расширению файла и другим компонентам.

Компоненты имени файла

Как видно из объявления `FLT_FILE_NAME_INFORMATION`, полное имя файла состоит из нескольких компонентов. Том — имя устройства, которому соответствует символическая ссылка «C:». На рис. 10.8 показана программа WinObj с символической ссылкой C: и ее целью (`\Device\HarddiskVolume3` на этой машине).

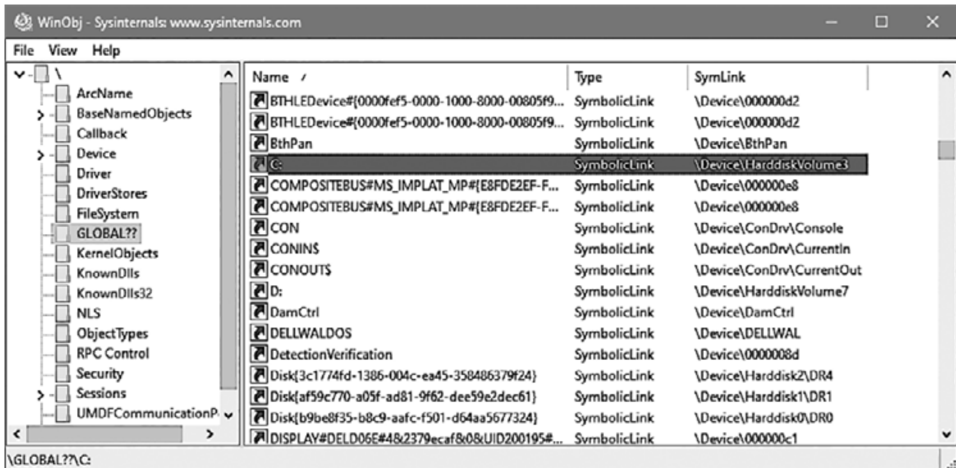


Рис. 10.8. Отображение символической ссылки в WinObj

Строка `share` для локальных файлов пуста (значение `Length` равно нулю). `ParentDir` присваивается только каталог; в нашем примере это будет строка `\mydir1\mydir2\` (без завершающего символа `\`). В поле `Extension` хранится расширение (`txt` в нашем примере). Поле `FinalComponent` содержит имя файла и имя потока данных (если используется поток данных, отличный от потока по умолчанию) — myfile.txt в нашем примере.

Компонент `Stream` заслуживает особого упоминания. Некоторые файловые системы (прежде всего NTFS) предоставляют возможность создания нескольких «потоков данных» (data streams) в одном файле. По сути это означает, что в одном «физическом» файле могут храниться сразу несколько файлов. Например, в NTFS то, что мы обычно считаем данными файла, на самом деле является лишь одним из потоков с именем `$DATA`, который считается потоком

по умолчанию. Однако вы можете создать/открыть другой поток, который хранится в том же файле. Такие программы, как «Проводник» Windows, не рассматривают эти потоки данных, и размеры альтернативных потоков не показываются и не возвращаются стандартными функциями API (такими, как `GetFileSize`). Имена потоков данных отделяются двоеточием от имени файла. Например, строка `myfile.txt:mystream` указывает на альтернативный поток данных `mystream` в файле с именем `myfile.txt`. Альтернативные потоки данных могут создаваться командным интерпретатором, как показывает следующий пример:

```
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. Есть ли там данные? Попытка воспользоваться командой `type` завершается неудачей:

```
C:\Temp>type hello.txt:mystream
The filename, directory name, or volume label syntax is incorrect.
```

Команда `type` не поддерживает имена потоков данных. Для вывода имени и размеров альтернативных потоков данных в файлах можно воспользоваться программой Streams.exe из пакета SysInternals. Результат выполнения команды для файла hello.txt:

```
C:\Temp>streams -nobanner hello.txt
C:\Temp\hello.txt:
        :mystream:$DATA 8
```

Содержимое альтернативного потока данных не выводится. Для просмотра (и возможно, экспортирования в другой файл) данных потока можно воспользоваться программой NtfsStreams из моего Github-репозитория AllTools.

На рис. 10.9 представлена программа NtfsStreams, в которой открывается файл hello.txt из предыдущего примера. В ней хорошо виден размер потока данных и содержащаяся в нем информация.

> Для потока данных указан тип $DATA (также существуют другие заранее определенные типы потоков данных). Нестандартные типы потоков данных обычно используются в точках повторного разбора данных (эта тема в книге не рассматривается).

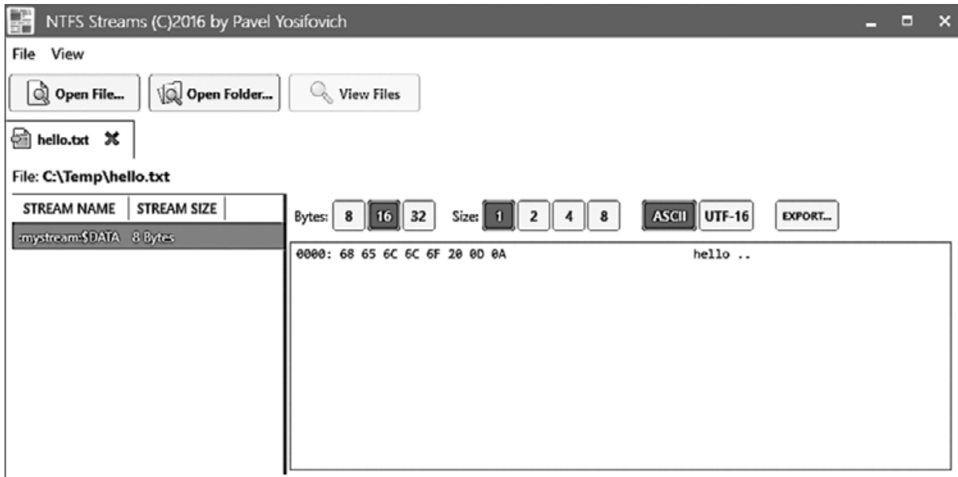


Рис. 10.9. Альтернативные потоки данных в NtfsStreams

Конечно, альтернативные потоки можно создавать на программном уровне — для этого следует указать имя потока после имени файла через двоеточие при вызове функции `CreateFile`. 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);
```

> Потоки данных могут удаляться обычным способом функцией `DeleteFile`, и их можно перебирать функциями `FindFirstStream` и `FileNextStream` (что и делают программы streams.exe и ntfsstreams.exe).

RAII-обертка FLT_FILE_NAME_INFORMATION

Как обсуждалось в предыдущем разделе, после вызова `FltGetFileNameInformation` должна быть вызвана парная функция `FltReleaseFileNameInformation`. Отсюда естественным образом появляется мысль о создании RAII-обертки, которая бы упрощала окружающий код и снижала риск ошибок. Одно из возможных объявлений такой обертки:

```
enum class FileNameOptions {
    Normalized = FLT_FILE_NAME_NORMALIZED,
```

```
    Opened = FLT_FILE_NAME_OPENED,
    Short = FLT_FILE_NAME_SHORT,

    QueryDefault = FLT_FILE_NAME_QUERY_DEFAULT,
    QueryCacheOnly = FLT_FILE_NAME_QUERY_CACHE_ONLY,
    QueryFileSystemOnly = FLT_FILE_NAME_QUERY_FILESYSTEM_ONLY,

    RequestFromCurrentProvider = FLT_FILE_NAME_REQUEST_FROM_CURRENT_PROVIDER,
    DoNotCache = FLT_FILE_NAME_DO_NOT_CACHE,
    AllowQueryOnReparse = FLT_FILE_NAME_ALLOW_QUERY_ON_REPARSE
};
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;
};
```

Функции, не определяемые как встроенные (non-inline), определяются 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);
}
```

Пример использования такой обертки:

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

Альтернативный драйвер Delete Protector

Создадим альтернативную версию драйвера Delete Protector, которая будет защищать от удаления файлы в некоторых каталогах (независимо от вызывающего процесса), вместо того чтобы принимать решение на основании вызывающего процесса или файла образа.

Конечно, эти два подхода можно объединить в одном драйвере или в нескольких драйверах с разными значениями `Altitude`.

Описанный в этом разделе драйвер DelProtect3 входит в число примеров для главы 10.

Сначала необходимо организовать управление каталогами, для которых включается защита (вместо имен файлов образов процессов, как в предыдущей версии драйвера). Здесь ситуация усложняется, потому что клиент пользовательского режима использует каталоги вида `c:\somedir` (то есть пути, основанные на символических ссылках). Как вы уже видели, драйвер получает реальные имена устройств вместо символических ссылок. А следовательно, имена в стиле DOS (как их иногда называют) необходимо преобразовать в имена в стиле NT (другое распространенное обозначение внутренних имен устройств).

По этой причине в списке защищенных каталогов каждый каталог присутствует в двух формах. Определение структуры выглядит так:

```
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;
        }
    }
};
```

Так как память для строк выделяется динамически, в какой-то момент их придется освободить. В приведенном коде добавляется метод `Free`, который освобождает внутренние строковые буферы. Выбор `UNICODE_STRING` вместо низкоуровневых строк C или даже строки постоянного размера отчасти произволен, но для потребностей драйвера такого решения должно быть достаточно. В данном случае я решил использовать объекты `UNICODE_STRING`, потому что сами строки могут выделяться динамически, а некоторые функции API работают с `UNICODE_STRING` напрямую.

Теперь мы можем сохранить массив этих структур и управлять им так же, как это делалось в предыдущем драйвере:

```
const int MaxDirectories = 32;

DirectoryEntry DirNames[MaxDirectories];
int DirNamesCount;
FastMutex DirNamesLock;
```

Смысл кодов управляющих операций ввода/вывода из предыдущего драйвера изменился — они должны добавлять и удалять каталоги, которые также представлены строками. Обновленные определения выглядят так:

```
#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)
```

Теперь необходимо реализовать операции добавления/удаления/очистки, начиная с добавления. Прежде всего необходимо проверить входную строку:

```
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) {
        // Слишком большая длина для каталога
        status = STATUS_INVALID_PARAMETER;
```

```
        break;
    }

    // Где-то должен находиться NULL-символ
    name[bufferLen / sizeof(WCHAR) - 1] = L'\0';

    auto dosNameLen = ::wcslen(name);
    if (dosNameLen < 3) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
```

Теперь, когда у нас имеется подходящий буфер, следует проверить, существует ли искомый элемент в массиве, и если существует, добавлять его повторно не нужно. Для выполнения поиска будет создана вспомогательная функция:

```
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 перебирает содержимое массива в поисках совпадения со входной строкой. Логический параметр указывает, какое имя следует искать в массиве — имя DOS или имя NT. Для проверки равенства используется функция `RtlEqualUnicodeString` с проверкой без учета регистра символов (последний аргумент `TRUE`). Функция возвращает индекс, по которому была найдена строка, или –1 при ее отсутствии. Обратите внимание: функция не захватывает никакую блокировку, поэтому сторона вызова должна вызвать функцию с необходимой синхронизацией.

Обработчик добавления каталога в список теперь должен провести поиск входной строки, а затем двигаться дальше, если строка будет найдена:

```
AutoLock locker(DirNamesLock);

UNICODE_STRING strName;
RtlInitUnicodeString(&strName, name);
if (FindDirectory(&strName, true) >= 0) {
    // Значение найдено, продолжить и вернуть признак успеха
    break;
}
```

После захвата быстрого мьютекса можно безопасно обращаться к массиву каталогов. Если строка не найдена, значит, функция получила новый каталог,

который нужно добавить в массив. Для начала убедимся в том, что размер массива еще не исчерпан:

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

На этой стадии необходимо перебрать массив и найти пустой слот (у которого указатель буфера DOS-строки содержит `NULL`). После этого можно добавить имя DOS и каким-то образом преобразовать его в имя NT, которое определенно пригодится позже.

```
for (int i = 0; i < MaxDirectories; i++) {
    if (DirNames[i].DosName.Buffer == nullptr) {
        // Оставить место для завершающего символа \ и NULL-завершителя
        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);

        // Присоединить символ \, если он отсутствует
        if (name[dosNameLen - 1] != L'\\')
            ::wcscat_s(buffer, dosNameLen + 2, L"\\");

        status = ConvertDosNameToNtName(buffer, &DirNames[i].NtName);

        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 получается достаточно прямолинейным, не считая разве что вызова `ConvertDosNameToNtName`. Это не встроенная функция — ее придется реализовать самостоятельно. Объявление выглядит так:

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

Как преобразовать имя DOS в имя NT? Так как «C:» и другие подобные имена являются символическими ссылками, один из методов заключается в том, чтобы

найти символическую ссылку и определить ее цель, которая представляет собой имя NT. Начнем с основных проверок:

```
ntName->Buffer = nullptr; // В случае неудачи
auto dosNameLen = ::wcslen(dosName);

if (dosNameLen < 3)
    return STATUS_BUFFER_TOO_SMALL;

// Убедиться в том, что буква диска присутствует
if (dosName[2] != L'\\' || dosName[1] != L':')
    return STATUS_INVALID_PARAMETER;
```

Каталог должен задаваться в форме `X:\...`: буква диска, двоеточие, обратный слеш и дальнейший путь. В нашем драйвере не поддерживаются ресурсы общего доступа (вида «`\myserver\myshare\mydir`»). Реализация их поддержки предлагается читателю для самостоятельной работы.

Теперь необходимо построить символическую ссылку, находящуюся в каталоге `\??\` диспетчера объектов. Для создания полной строки можно воспользоваться функциями строковых операций. В следующем фрагменте кода будет использоваться тип `kstring`, который является оберткой для строк (концептуальным аналогом стандартного типа C++ `std::wstring`). Реализация этого типа более подробно рассматривается в главе 11.

> Вы можете изменить приведенный ниже код и использовать другие строковые функции для получения того же результата. Пусть это станет очередным упражнением для читателя.

Начнем с базового каталога символической ссылки и добавим полученную букву диска:

```
kstring symLink(L"\\??\\");
symLink.Append(dosName, 2); // Буква диска и двоеточие
```

Откроем символическую ссылку функцией `ZwOpenSymbolicLinkObject`. Для этого следует подготовить структуру `OBJECT_ATTRIBUTES`, общую для многих «открывающих» функций API, требующих передачи некоторого имени:

```
UNICODE_STRING symLinkFull;
symLink.GetUnicodeString(&symLinkFull);

OBJECT_ATTRIBUTES symLinkAttr;
InitializeObjectAttributes(&symLinkAttr, &symLinkFull,
    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);
```

`GetUnicodeString` — вспомогательная функция `kstring`, которая инициализирует `UNICODE_STRING` для заданного объекта `kstring`. Это необходимо, потому что

OBJECT_ATTRIBUTES требует использования UNICODE_STRING для имени. Инициализация OBJECT_ATTRIBUTES осуществляется макросом InitializeObjectAttributes, который должен получать следующие аргументы (в указанном порядке):

- Указатель на инициализируемую структуру OBJECT_ATTRIBUTES.
- Имя объекта.
- Набор флагов. В данном случае возвращаемый дескриптор должен быть дескриптором ядра, а поиск должен выполняться без учета регистра символов.
- Необязательный дескриптор корневого каталога на случай, если имя является относительным, а не абсолютным (NULL в данном случае).
- Необязательный дескриптор безопасности, применяемый к объекту (NULL в данном случае).

После того как структура будет инициализирована, все готово для вызова ZwOpenSymbolicLinkObject:

```
HANDLE hSymLink = nullptr;
auto status = STATUS_SUCCESS;

do {
    // Открытие символической ссылки
    status = ZwOpenSymbolicLinkObject(&hSymLink, GENERIC_READ, &symLinkAttr);
    if (!NT_SUCCESS(status))
        break;
```

Мы воспользуемся хорошо знакомой конструкцией do / while(false) для завершения дескриптора, если он действителен. ZwOpenSymbolicLinkObject получает выходной дескриптор HANDLE, маску доступа (GENERIC_READ означает, что мы собираемся читать информацию) и атрибуты, подготовленные ранее. Конечно, этот вызов может завершиться неудачей (например, если указанная буква диска не существует).

Если вызов завершается успехом, необходимо прочитать цель, на которую указывает объект символической ссылки. Эта информация может быть получена при помощи функции ZwQuerySymbolicLinkObject. Необходимо подготовить структуру UNICODE_STRING с размером, достаточным для хранения результата (которая также является выходным параметром функции преобразования):

```
    USHORT maxLen = 1024; // Выбрано произвольно
    ntName->Buffer = (WCHAR*)ExAllocatePool(PagedPool, maxLen);
    if (!ntName->Buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        break;
    }
    ntName->MaximumLength = maxLen;
```

```
    // Чтение цели символической ссылки
    status = ZwQuerySymbolicLinkObject(hSymLink, ntName, nullptr);
    if (!NT_SUCCESS(status))
        break;
} while (false);
```

После выхода из блока `do/while` необходимо освободить выделенный буфер, если что-то пошло не так. В противном случае оставшуюся часть входного каталога можно присоединить к целевому имени NT:

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

Наконец, необходимо закрыть дескриптор символической ссылки, если он был открыт успешно:

```
    if (hSymLink)
        ZwClose(hSymLink);

    return status;
}
```

При удалении защищенного каталога мы выполняем аналогичные проверки с защищенным путем, а затем ищем его по имени DOS. Если каталог будет найден, он удаляется 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;
```

Операция очистки реализуется очень просто. Я оставлю ее читателю для самостоятельной работы (решение можно найти в исходном коде проекта).

Обработка информации перед созданием и назначением информации

При наличии описанной выше инфраструктуры можно обратиться к реализации обратных вызовов перед операциями, предотвращающих удаление файлов в защищенных каталогах независимо от вызывающего процесса. В обоих обратных вызовах необходимо получить полное имя удаляемого файла и проверить наличие каталога в массиве каталогов. Мы создадим для этой цели вспомогательную функцию, которая объявляется следующим образом:

```
bool IsDeleteAllowed(_In_ PFLT_CALLBACK_DATA Data);
```

Так как нужный файл упакован в структуру `FLT_CALLBACK_DATA`, это все, что нам понадобится. Прежде всего необходимо получить полное имя (в этом коде мы не будем использовать обертку, представленную ранее, чтобы вызовы функций API стали более наглядными):

```
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 имени файла и разбираем ее, так как нам нужен только том и родительский каталог (и ресурс общего доступа, если он поддерживается). Нам понадобится структура `UNICODE_STRING`, объединяющая три компонента:

```
// Конкатенация тома, ресурса общего доступа и каталога
UNICODE_STRING path;
path.Length = path.MaximumLength =
    nameInfo->Volume.Length + nameInfo->Share.Length
                            + nameInfo->ParentDir. Length;
path.Buffer = nameInfo->Volume.Buffer;
```

Так как полный путь к файлу хранится в непрерывной области памяти, указатель на буфер начинается с первого компонента (том), и длина должна вычисляться соответствующим образом. Остается сделать еще один шаг: 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);
```

Далее остается лишь освободить информацию имени файла:

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

Вернемся к обратным вызовам перед операциями. Начнем с обратного вызова перед созданием:

```
_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) {
        // Операция удаления
        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;
}
```

Обратный вызов перед назначением информации выглядит аналогично:

```
_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) {
        // Не является операцией удаления
        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 был обновлен для отправки кодов управляющих операций, хотя код очень похож на приводившийся выше, поскольку строки для добавления/удаления отправляются точно так же, как и прежде (проект DelProtectConfig3 в исходном коде). Примеры 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 сценариях бывает удобно связывать дополнительные данные с такими сущностями файловой системы, как тома и файлы. Диспетчер фильтров предоставляет эту возможность в виде *контекстов*. Контекст представляет собой структуру данных, предоставляемую драйвером мини-фильтров, которая может задаваться и читаться для любого объекта файловой системы. Эти контексты остаются связанными с объектами, для которых они назначаются, на все время существования этих объектов.

Чтобы использовать контексты, драйвер должен заранее объявить, какие контексты и для каких типов объектов ему могут понадобиться. Это делается при помощи структуры регистрации `FLT_REGISTRATION`. Поле `ContextRegistration` может указывать на массив структур `FLT_CONTEXT_REGISTRATION`, каждая из которых определяет информацию для одного контекста. Объявление структуры `FLT_CONTEXT_REGISTRATION` выглядит 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;
```

Описание полей структуры:

`ContextType` определяет тип объекта, к которому присоединяется контекст. `FLT_CONTEXT_TYPE` определяется с типом `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 этих определений, контекст может быть присоединен к тому, экземпляру фильтра, файлу, потоку данных, дескриптору потока данных, транзакции или секции (в Windows 8 и выше). Последнее (сторожевое) значение — признак конца списка определений контекстов. Врезка «Типы контекстов» содержит дополнительную информацию о различных типах контекстов.

Размер контекста может быть фиксированным или переменным. Если нужен фиксированный размер, он задается в поле `Size` структуры `FLT_CONTEXT_REGISTRATION`. Для контекста переменного размера драйвер задает специальное значение `FLT_VARIABLE_SIZED_CONTEXTS` (–1). Использование контекстов фиксированного размера более эффективно, потому что диспетчер фильтров может использовать списки хранения данных (lookaside lists) для управления выделением и освобождением памяти (за дополнительной информацией о списках хранения данных обращайтесь к документации WDK).

Тег пула задается в поле `PoolTag` структуры `FLT_CONTEXT_REGISTRATION`. Этот тег используется диспетчером фильтров при фактическом выделении контекста. Следующие два поля содержат необязательные обратные вызовы, в которых драйвер предоставляет функции выделения и освобождения. Если эти поля отличны от `NULL`, то поля `PoolTag` и `Size` не имеют смысла и не используются.

Пример построения массива структур регистрации контекстов:

```
struct FileContext {
    //...
};

const FLT_CONTEXT_REGISTRATION ContextRegistration[] = {
    { FLT_FILE_CONTEXT, 0, nullptr, sizeof(FileContext), 'torP',
        nullptr, nullptr, nullptr },
    { FLT_CONTEXT_END }
};
```

ТИПЫ КОНТЕКСТОВ

Диспетчер фильтров поддерживает несколько типов контекстов:

- Контексты томов присоединяются к томам, например дисковым разделам (C:, D: и т. д.).
- Контексты экземпляров присоединяются к экземплярам фильтров. Мини-фильтр может иметь несколько работающих экземпляров, каждый из которых присоединен к отдельному тому.
- Контексты файлов могут присоединяться к файлам вообще (а не к конкретному потоку данных в файле).
- Контексты потоков данных могут присоединяться к потокам данных в файлах, поддерживаемым некоторыми файловыми системами (в частности, NTFS). Файловые системы, поддерживающие один поток данных на файл (например, FAT), интерпретируют контексты потоков данных как контексты файлов.
- Контексты дескрипторов потоков данных могут присоединяться к потокам данных на уровне `FILE_OBJECT`.
- Контексты транзакций могут присоединяться к незавершенным транзакциям. Файловая система NTFS поддерживает транзакции, и этот факт позволяет связывать контексты с работающими транзакциями.
- Контексты секций могут присоединяться к секциям (объектам отображения файлов), созданным функцией `FltCreateSectionForDataScan` (тема выходит за рамки книги).

Не все типы контекстов поддерживаются во всех файловых системах. Диспетчер фильтров предоставляет функции API для динамического получения информации (для некоторых типов контекстов): `FltSupportsFileContexts`, `FltSupportsFileContextsEx`, `FltSupportsStreamContexts` и т. д.

Управление контекстами

Чтобы использовать контекст, драйвер должен сначала выделить память вызовом функции `FltAllocateContext`, которая определяется следующим образом:

```
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` содержит непрозрачный указатель, возвращаемый вызовом `FltRegisterFilter`, но также доступный в структуре `FLT_RELATED_OBJECTS`, передаваемой всем обратным вызовам. `ContextType` — один из поддерживаемых контекстных макросов, упоминавшихся ранее (например, `FLT_FILE_CONTEXT`). `ContextSize` — запрашиваемый размер контекста в байтах (должен быть больше нуля). Поле `PoolType` может содержать `PagedPool` или `NonPagedPool` в зависимости от того, на каком уровне IRQL драйвер планирует обращаться к контексту (для контекстов томов должно быть задано значение `NonPagedPool`). Наконец, в поле `ReturnedContext` хранится возвращаемый выделенный контекст; `PFLT_CONTEXT` определяется как тип `PVOID`.

После того как контекст будет выделен, драйвер может сохранить в этом буфере данных все, что пожелает. Затем он должен присоединить контекст к объекту (для чего, собственно, контекст и создавался) при помощи одной из функций `FltSet`*Xxx*`Context`, где *Xxx* — один из вариантов: `File`, `Instance`, `Volume`, `Stream`, `StreamHandle` или `Transaction`. Единственным исключением является контекст секции, который создается вызовом `FltCreateSectionForDataScan`. Каждая из функций `FltSet`*Xxx*`Context` имеет одну и ту же общую схему, которая приводится ниже для случая `File`:

```
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 получает необходимые параметры для имеющегося контекста. В данном случае (`File`) это экземпляр (необходимый в любой функции назначения контекста) и объект файла, представляющий файл, с которым связывается контекст. Параметр `Operation` может быть равен `FLT_SET_CONTEXT_REPLACE_IF_EXISTS` или `FLT_SET_CONTEXT_KEEP_IF_EXISTS`.

`NewContext` содержит назначаемый контекст, а `OldContext` — необязательный параметр, который может использоваться для получения предыдущего контекста с операцией `FLT_SET_CONTEXT_REPLACE_IF_EXISTS`.

На контексты действует механизм подсчета ссылок. Операции выделения контекста (`FltAllocateContext`) и назначения контекста увеличивают его счетчик ссылок. Парная функция `FltReleaseContext` должна вызываться соответствующее количество раз, чтобы предотвратить утечку контекста. Хотя функция удаления контекста существует (`FltDeleteContext`), обычно она не нужна, потому что диспетчер фильтров уничтожает контекст при уничтожении объекта файловой системы, с которым этот контекст связан.

Будьте внимательны при управлении контекстами! Иначе может оказаться, что драйвер не может быть выгружен из-за того, что контекст с положительным счетчиком ссылок продолжает существовать, а объект файловой системы, к которому он присоединен (например, файл или том), еще не был удален. Очевидно, для работы с контекстами могла бы пригодиться RAII-обертка.

В типичном сценарии вы выделяете память для контекста, заполняете его, связываете с соответствующим объектом и вызываете `FltReleaseContext`, в результате чего счетчик ссылок контекста остается равным 1. Практическое применение контекстов продемонстрировано в разделе «Драйвер File Backup» этой главы.

После того как контекст будет связан с объектом, другие обратные вызовы могут захватить этот контекст. Семейство функций `get` предоставляет доступ к соответствующему контексту (имена всех этих функций строятся по схеме `FltGet`*Xxx*`Context`, где *Xxx* — `File`, `Instance`, `Volume`, `Stream`, `StreamHandle`, `Transaction` или `Section`). Функции `get` увеличивают счетчик ссылок контекста, поэтому после завершения работы с контекстом необходимо вызвать `FltReleaseContext`.

Инициирование запросов ввода/вывода

Мини-фильтры файловой системы иногда должны инициировать собственные операции ввода/вывода. Обычно код режима ядра использует такие функции, как `ZwCreateFile`, для открытия дескриптора файла, а затем выдает операции ввода/вывода при помощи таких функций, как `ZwReadFile`, `ZwWriteFile`, `ZwDeviceIoControlFile` и некоторых других. Мини-фильтры обычно не используют `ZwCreateFile`, если им требуется инициировать операцию ввода/вывода из одного из обратных вызовов диспетчера фильтров. Это объясняется тем, что операция ввода/вывода пройдет от верхнего фильтра вниз до самой файловой системы и встретит на пути текущий мини-фильтр!

Возникает ситуация повторного входа (reentrancy), которая может создать проблемы, если драйвер не примет меры предосторожности. Также страдает быстродействие, потому что запрос должен пройти весь стек фильтров файловой системы.

Вместо этого мини-фильтры используют функции диспетчера фильтров для выдачи операций ввода/вывода, которые отправляются следующему фильтру более низкого уровня по направлению к файловой системе; тем самым предотвращается повторный вход и возможные потери для быстродействия. Такие функции API начинаются с префикса `Flt` и на концептуальном уровне аналогичны разновидностям `Zw`. Чаще всего используется функция `FltCreateFile` (и ее расширенные аналоги `FltCreateFileEx` и `FltCreateFileEx2`). Прототип `FltCreateFile` выглядит так:

```
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);
```

Как видите, эта функция имеет много параметров. К счастью, понять их смысл несложно, но эти параметры необходимо задать правильно, иначе вызов завершится с невразумительным статусом.

Как видно из объявления, первый аргумент содержит непрозрачный адрес фильтра, который используется в качестве базового уровня для операций ввода/вывода через полученный дескриптор файла. Главное возвращаемое значение — дескриптор открытого файла `FileHandle` в случае успеха. Мы не будем рассматривать все возможные параметры (обращайтесь к документации WDK), но функция будет использована в следующем разделе.

> Расширенная функция `FltCreateFileEx` имеет дополнительный выходной параметр — указатель на объект `FILE_OBJECT`, созданный функцией. `FltCreateFileEx2` имеет дополнительный входной параметр типа `IO_DRIVER_CREATE_CONTEXT`, используемый для передачи дополнительной информации файловой системе (за дополнительной информацией обращайтесь к документации WDK).

С возвращенным дескриптором драйвер может вызвать стандартные функции API ввода/вывода, такие как `ZwReadFile`, `ZwWriteFile` и т. д. Операция все равно будет обращена только к нижним уровням. Также драйвер может использовать структуру `FILE_OBJECT`, возвращенную из `FltCreateFileEx` или `FltCreateFileEx2`, с такими функциями, как `FltReadFile` и `FltWriteFile` (последней требуется объект файла вместо дескриптора).

После завершения операции для возвращенного дескриптора должна быть вызвана функция `FltClose`. Если, кроме этого, был возвращен объект файла, также необходимо уменьшить его счетчик ссылок вызовом `ObDereferenceObject` для предотвращения утечки.

Функция `FltClose` в действительности просто вызывает `ZwClose`; она существует просто ради логической целостности.

Драйвер File Backup

Пришло время применить на практике полученные знания, а именно использование контекстов и операций ввода/вывода из драйвера мини-фильтра. Драйвер, который мы построим, обеспечивает автоматическое создание резервной копии файлов каждый раз, когда эти файлы открываются для записи, непосредственно перед записью. Это позволяет вернуться к предыдущему состоянию файла в случае необходимости. По сути, у любого файла в любой момент времени существует одна резервная копия.

Главный вопрос: где должна храниться эта резервная копия? Например, можно создать каталог backup в каталоге этого файла или же создать корневой каталог для всех резервных копий и создавать резервную копию в той же структуре папок, что и от исходного файла, но начиная от корневого каталога резервных копий (драйвер даже может скрыть этот каталог). Оба варианта приемлемы, но в этой демонстрационной программе будет использован другой вариант: резервная копия файла будет храниться в самом файле в альтернативном потоке данных NTFS. В сущности, файл будет содержать свою собственную резервную копию. Тогда в случае необходимости можно поменять местами контексты альтернативного потока данных и потока данных по умолчанию, фактически восстанавливая файл в его предыдущем состоянии.

Начнем с шаблона проекта мини-фильтра файловой системы, который использовался в предыдущих драйверах. Драйверу будет присвоено имя FileBackup. Затем необходимо изменить INF-файл как обычно. Некоторые части, которые были изменены:

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

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

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

Файл FileBackup.c будет переименован в FileBackup.cpp для поддержки кода C++.

Так как мы будем работать с альтернативными потоками, использоваться может только NTFS, потому что это единственная «стандартная» файловая система семейства Windows, поддерживающая альтернативные файловые потоки. Это означает, что драйвер не должен присоединяться к томам, не использующим NTFS. Драйвер должен изменить стандартную реализацию обратного вызова «настройки экземпляра», уже созданную шаблоном проекта. Полный код функции с дополнительной проверкой NTFS и отказом от использования файловых систем, отличных от 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 присоединение к указанному тому.

Затем необходимо зарегистрироваться для получения нужных запросов. Драйвер должен перехватывать операции записи, поэтому понадобится обратный

вызов перед операцией для запроса IRP_MJ_WRITE. Также необходимо отслеживать состояние с использованием файлового контекста. Возможно, драйверу также понадобится обработка после операции создания и операции очистки (IRP_MJ_CLEANUP). Позднее вы увидите, для чего это нужно. А пока с учетом этих ограничений создадим структуру регистрации обратных вызовов:

```
#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 }
};
```

Также понадобится контекст для хранения информации о том, выполнялась ли ранее операция записи с конкретным открытым файлом. Определим структуру контекста:

```
struct FileContext {
    Mutex Lock;
    UNICODE_STRING FileName;
    BOOLEAN Written;
};
```

В структуре будет храниться само имя файла (будет проще иметь его наготове при создании резервной копии), мьютекс для целей синхронизации и логический флаг, указывающий, выполнялась ли ранее с этим файлом операция создания резервной копии. Практическое использование контекста станет более понятным, когда мы начнем вызывать реализацию обратного вызова.

Контекст необходимо зарегистрировать в массиве контекстов:

```
const FLT_CONTEXT_REGISTRATION Contexts[] = {
    { FLT_FILE_CONTEXT, 0, nullptr, sizeof(FileContext), DRIVER_CONTEXT_TAG },
    { FLT_CONTEXT_END }
};
```

Ссылка на массив хранится в полной структуре регистрации:

```
CONST FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),         // Размер
    FLT_REGISTRATION_VERSION,         // Версия
    0,                                // Флаги

    Contexts,                         // Контекст
    Callbacks,                        // Обратные вызовы операций
```

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

Итак, все структуры готовы, и мы можем перейти к реализации обратного вызова.

Обратный вызов после создания

Зачем может понадобиться обратный вызов после создания? На самом деле драйвер можно написать и без него, но этот обратный вызов поможет продемонстрировать некоторые возможности, которые нам еще не встречались. В обратном вызове после создания мы будем выделять контекст для файлов, которые нас интересуют. Например, файлы, которые не были открыты для записи, не представляют интереса для драйвера.

Почему мы используем обратный вызов после операции вместо обратного вызова перед операцией? Если операция открытия файла завершится неудачей в обратном вызове перед созданием в другом драйвере, нас это не интересует. Только если файл будет открыт успешно, наш драйвер должен продолжить проверку файла. Начало реализации:

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

Затем извлекаются параметры операции создания:

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

Нас интересуют только файлы, открытые для записи, не из режима ядра и не являющиеся новыми (так как новые файлы не требуют резервного копирования). Соответствующие проверки выглядят так:

```
if (Data->RequestorMode == KernelMode
    || (params.SecurityContext->DesiredAccess & FILE_WRITE_DATA) == 0
    || Data->IoStatus.Information == FILE_DOES_NOT_EXIST) {
    // Вызов из режима ядра, не для записи и не новый файл - пропустить
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

За дополнительной информацией о коде, приведенном выше, обращайтесь к документации структуры `FLT_PARAMETERS` для `IRP_MJ_CREATE`.

Такие проверки очень важны, так как они избавляют драйвер от значительного объема лишней работы. Драйвер всегда должен стремиться к тому, чтобы выполнять как можно меньше работы, чтобы снизить свое влияние на быстродействие.

Итак, мы имеем файл, который представляет для нас интерес. Необходимо подготовить объект контекста, который будет присоединен к файлу. Контекст понадобится позднее, когда мы займемся обратным вызовом перед записью. Начнем с извлечения имени файла. Для этого драйвер должен вызвать стандартную функцию FltGetFileNameInformation. Чтобы немного упростить задачу и снизить риск ошибки, мы воспользуемся RAII-оберткой, представленной 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;
```

На следующем шаге нужно решить, нужно ли создавать резервные копии всех файлов или только файлов, находящихся в определенных каталогах. Для гибкости будет выбран второй вариант. Создадим вспомогательную функцию IsBackupDirectory, которая должна возвращать true для нужных каталогов. Ниже приведена простая реализация, которая возвращает true для любого каталога с именем \pictures\ или \documents\:

```
bool IsBackupDirectory(_In_ PCUNICODE_STRING directory) {
    // нет определенной версии 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 получает только имя каталога (извлекается вызовом `FltParseFileNameInformation`), в котором она должна искать указанные подстроки. К сожалению, это не так просто. Для этого естественно использовать функцию `wcsstr`, которая сканирует строку в поисках определенной подстроки, но тут возникают две проблемы:

- Функция учитывает регистр символов, что может быть неудобно при работе с файлами или каталогами.
- Функция ожидает, что строка, в которой производится поиск, завершается NULL-символом, что не гарантировано для `UNICODE_STRING`.

Из-за этих нюансов код выделяет собственный строковый буфер, копирует имя каталога и преобразует строку к нижнему регистру (`_wcslwr`), прежде чем использовать `wcsstr` для поиска `\pictures\` и `\documents\`.

> Конечно, в драйвере коммерческого уровня эти жестко запрограммированные строки должны поступать от программы настройки, реестра и т. д. Реализация этой возможности предлагается читателю для самостоятельной работы.

В обратном вызове после создания мы вызываем `IsBackupDirectory` и возвращаем управление, если было получено значение `false`:

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

Осталось выполнить еще одну проверку. Если файл открывается для потока данных, отличного от потока по умолчанию, создавать резервную копию не нужно. Резервная копия создается только для потока данных по умолчанию:

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

Наконец, можно выделить память для контекста 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);

// Инициализировать мьютекс
context->Lock.Init();
```

Этот код заслуживает некоторых пояснений. `FltAllocateContext` выделяет для контекста память необходимого размера и возвращает указатель на выделенную память. `PFLT_CONTEXT` — это всего лишь указатель `void*`; его можно преобразовать к любому нужному типу. Возвращаемая память контекста не заполнена нулями, поэтому все поля необходимо инициализировать.

Зачем вообще нужен этот контекст? Типичный клиент открывает файл для записи, после чего вызывает `WriteFile` — возможно, несколько раз. Перед первым вызовом `WriteFile` драйвер должен создать резервную копию существующего содержимого файла. Становится понятно, для чего нужен флаг `Written`, — чтобы резервная копия создавалась только перед первой операцией записи. В исходном состоянии флаг содержит `FALSE`, а после первой операции записи переходит в состояние `TRUE`. Эта последовательность событий изображена на рис. 10.10.

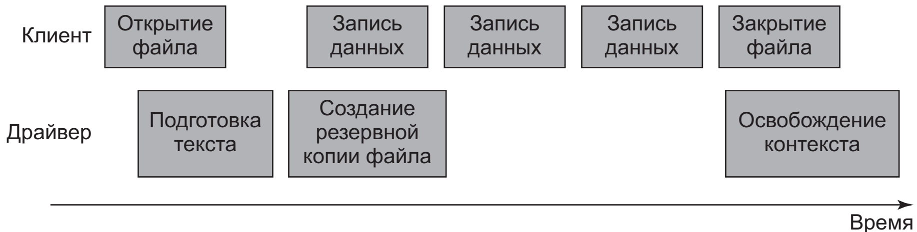


Рис. 10.10. Операции клиента и драйвера для типичной последовательности действий при записи

Затем необходимо выделить память для хранения полного имени файла; оно понадобится позднее, когда мы будем создавать резервную копию файла. С технической точки зрения можно вызвать `FltGetFileNameInformation` в момент создания резервной копии, но поскольку вызов функции может завершиться неудачей в некоторых ситуациях, лучше получить имя файла сразу и использовать его позднее, чтобы сделать драйвер более устойчивым к ошибкам.

Последнее поле в нашем контексте содержит мьютекс. Нам понадобится некоторая форма синхронизации для нетипичного, но возможного случая, в котором сразу несколько потоков в одном клиентском процессе выполняют запись в один файл приблизительно в одно время. В таком случае необходимо позаботиться о том, чтобы создавалась только одна резервная копия; в противном случае копия может оказаться поврежденной. Во всех предыдущих примерах,

в которых требовалась синхронизация, мы использовали быстрый мьютекс, но здесь используется стандартный мьютекс. Почему? Это связано с операциями, которые будут вызываться драйвером при создании резервной копии файла: такие функции ввода/вывода, как ZwWriteFile и ZwReadFile, могут вызываться только на уровне IRQL PASSIVE_LEVEL (0). Захваченный быстрый мьютекс повышает IRQL до APC_LEVEL (1), что породит взаимную блокировку при использовании функций API ввода/вывода.

Здесь используется класс Mutex, приведенный в главе 6, который будет использоваться с RAII-классом AutoLock, уже неоднократно использовавшимся в предыдущих главах.

Контекст успешно инициализирован, поэтому теперь нужно присоединить его к файлу вызовом FltSetFileContext, освободить контекст и, наконец, вернуть управление из обратного вызова после создания:

```
    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;
}
```

Функции API назначения контекста позволяют сохранить существующий контекст (если он существует) или заменить его (если он существует). В данном случае мы решили сохранить существующий контекст в редком случае, в котором две разные стороны открывают один файл для записи и одна успевает быстрее задать контекст. Если контекст уже присутствовал, возвращается статус STATUS_FLT_CONTEXT_ALREADY_DEFINED, который является признаком ошибки, а при возвращении признака ошибки драйвер должен освободить строковый буфер, выделенный ранее.

Наконец, вызывается функция FltReleaseContext, которая (если все идет нормально) устанавливает внутренний счетчик ссылок контекста равным 1 (+1 для выделения памяти, +1 для назначения, –1 для освобождения). Если контекст назначить не удалось, то он будет полностью освобожден.

Обратный вызов перед записью

Задача обратного вызова перед записью — создание копии данных файла непосредственно перед тем, как будет разрешено выполнение фактической операции

записи; вот почему здесь нужен обратный вызов перед операцией, в противном случае в обратном вызове после операции запись уже будет завершена.

Начнем с получения контекста файла. Если он не существует, значит, наша функция обратного вызова после создания посчитала, что файл не представляет интереса и 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);

    // Получить контекст файла, если он существует
    FileContext* context;

    auto status = FltGetFileContext(FltObjects->Instance,
        FltObjects->FileObject, (PFLT_CONTEXT*)&context);
    if (!NT_SUCCESS(status) || context == nullptr) {
        // Контекста нет, продолжить нормальное выполнение
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
    }
```

После того как контекст будет получен, необходимо создать копию данных файла только один раз — перед первой операцией записи. Сначала мы захватываем мьютекс и проверяем флаг `Written` из контекста. Если флаг равен `false`, значит, 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` играет ключевую роль в работе этой схемы. Можно подумать, что создание копии сводится к вызову одной функции API; к сожалению, это не так. В режиме ядра не существует функции `CopyFile`. Функция API пользовательского режима `CopyFile` — нетривиальная функция, которая выполняет довольно серьезную работу для реализации копирования. Одной из составляющих этой работы является чтение байтов

из исходного файла и запись их в приемный файл. Тем не менее в общем случае этого недостаточно. Во-первых, файл может содержать несколько потоков данных для копирования (в случае NTFS). Во-вторых, остается вопрос с дескриптором безопасности из исходного файла, который тоже необходимо скопировать в некоторых случаях (за подробностями обращайтесь к документации `CopyFile`).

Следовательно, нам придется создавать собственную операцию копирования файлов. К счастью, при этом достаточно скопировать один файловый поток данных — поток по умолчанию копируется в другой поток в том же физическом файле. Начало 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;
```

В выбранном нами решении будут открываться два дескриптора: один (исходный) связан с исходным файлом (с потоком по умолчанию для резервного копирования), а другой (целевой) — с потоком резервной копии. Затем мы будем читать данные из источника и записывать их в целевой поток. На концептуальном уровне все выглядит просто, но, как это часто бывает в программировании режима ядра, дьявол кроется в деталях.

Начнем с получения размера файла. Размер файла может быть нулевым, в этом случае ничего копировать не нужно:

```
LARGE_INTEGER fileSize;
status = FsRtlGetFileSize(FltObjects->FileObject, &fileSize);
if (!NT_SUCCESS(status) || fileSize.QuadPart == 0)
    return status;
```

Функцию API `FsRtlGetFileSize` рекомендуется использовать в тех случаях, когда нужно получить размер файла с заданным указателем на `FILE_OBJECT`. Также можно воспользоваться функцией `ZwQueryInformationFile` для получения размера файла (также функция может вернуть много других видов информации), но для этого нужен файловый дескриптор, а в некоторых ситуациях вызов может привести к взаимной блокировке.

Теперь можно открыть исходный файл вызовом `FltCreateFile`. Важно не использовать `ZwCreateFile`, чтобы запросы ввода/вывода отправлялись драйверу, находящемуся на более низком уровне, а не на вершину стека драйверов файловой системы. Функция `FltCreateFile` имеет много параметров, поэтому выглядит он устрашающе:

```
do {
    // Открыть исходный файл
    OBJECT_ATTRIBUTES sourceFileAttr;
    InitializeObjectAttributes(&sourceFileAttr, FileName,
        OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);

    status = FltCreateFile(
        FltObjects->Filter,            // Объект фильтра
        FltObjects->Instance,          // Экземпляр фильтра
        &hSourceFile,                  // Исходный дескриптор
        FILE_READ_DATA | SYNCHRONIZE,   // Маска доступа
        &sourceFileAttr,               // Атрибуты объекта
        &ioStatus,                     // Итоговый статус
        nullptr, FILE_ATTRIBUTE_NORMAL, // Размер выделения, атрибуты файла
        FILE_SHARE_READ | FILE_SHARE_WRITE, // Флаги ресурса общего доступа
        FILE_OPEN,                     // Режим создания
        FILE_SYNCHRONOUS_IO_NONALERT,   // Параметры создания (синхр. ввод/вывод)
        nullptr, 0,                    // Расширенные атрибуты, их длина
        IO_IGNORE_SHARE_ACCESS_CHECK);  // Флаги

if (!NT_SUCCESS(status))
    break;
```

Прежде чем вызывать `FltCreateFile` (как и любую другую функцию API, которой требуется имя), необходимо инициализировать структуру `OBJECT_ATTRIBUTES` именем файла, переданным `BackupFile`. Это файловый поток данных по умолчанию, который должен быть изменен операцией записи; именно поэтому для него создается резервная копия.

Важнейшие аргументы функции:

- Объекты `Filter` и `Instance`, предоставляющие необходимую информацию для передачи вызова фильтру следующего нижнего уровня (или файловой системы) — вместо вершины стека файловой системы.
- `hSourceFile` — возвращаемый дескриптор.
- Маска доступа, содержащая флаги `FILE_READ_DATA` и `SYNCHRONIZE`.
- Режим создания — в данном случае указывает, что файл должен существовать (`FILE_OPEN`).
- Параметры создания — содержат флаг `FILE_SYNCHRONOUS_IO_NONALERT`, обозначающий синхронные операции через возвращенный дескриптор файла. Для работы синхронной операции необходим флаг маски доступа `SYNCHRONIZE`.
- Флаг `IO_IGNORE_SHARE_ACCESS_CHECK` играет важную роль, потому что соответствующий файл был уже открыт клиентом, который, скорее всего, открыл его с запретом общего доступа. По этой причине мы приказываем файловой системе игнорировать проверки общего доступа для этого вызова.

> Чтобы лучше понять смысл различных параметров этой функции, обращайтесь к документации `FltCreateFile`.

Затем необходимо открыть или создать поток данных резервной копии в том же файле. Присвоим потоку имя `:backup` и воспользуемся другим вызовом `FltCreateFile` для получения дескриптора 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,             // Объект фильтра
    FltObjects->Instance,           // Экземпляр фильтра
    &hTargetFile,                   // Целевой дескриптор
    GENERIC_WRITE | SYNCHRONIZE,    // Маска доступа
    &targetFileAttr,                // Атрибуты объекта
    &ioStatus,                      // Итоговый статус
    nullptr, FILE_ATTRIBUTE_NORMAL, // Размер выделения, атрибуты файла
    0,                              // Флаги общего доступа
    FILE_OVERWRITE_IF,              // Режим создания
    FILE_SYNCHRONOUS_IO_NONALERT,   // Параметры создания (синхр. ввод/вывод)
    nullptr, 0, 0);                 // Расширенные атрибуты, их длина, флаги

ExFreePool(targetFileName.Buffer);

if (!NT_SUCCESS(status))
    break;
```

Имя файла строится конкатенацией базового имени файла и имени потока данных резервной копии. Файл открывается для записи (`GENERIC_WRITE`) с замещением всех имеющихся данных (`FILE_OVERWRITE_IF`).

Имея эти два дескриптора, можно начать с источника и записывать в целевой файл. Простое решение — выделение буфера с размером файла и выполнение всей работы одной операции чтения и одной операции записи. Тем не менее такое решение может создать проблемы при очень большом размере файла, а попытка выделения памяти может завершиться неудачей.

Также существует риск создания резервной копии очень большого файла, возможно, с большими затратами дискового пространства. Вероятно, в таком драйвере для очень больших файлов резервного копирования следует избегать (например, эта возможность может настраиваться в реестре) или отказаться от резервного копирования, если свободное дисковое пространство падает ниже определенного порога (который тоже может настраиваться). Эта возможность предоставляется читателю для самостоятельной реализации.

Проблема решается выделением относительно небольшого буфера и простым перебором его содержимого, пока не будут скопированы все фрагменты файла. Этот подход будет использован в нашем решении. Начнем с выделения памяти для буфера:

```
ULONG size = 1 << 21;     // 2 Мбайт
buffer = ExAllocatePoolWithTag(PagedPool, size, DRIVER_TAG);
if (!buffer) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
```

А теперь собственно цикл — мы будем использовать смещения, но можно обойтись и без них, потому что мы работаем с файлами синхронно, а каждый объект файла отслеживает текущую позицию в файле и увеличивает ее после каждой операции.

```
LARGE_INTEGER offset = { 0 };      // Чтение
LARGE_INTEGER writeOffset = { 0 }; // Запись

ULONG bytes;
auto saveSize = fileSize;
while (fileSize.QuadPart > 0) {
    status = ZwReadFile(
        hSourceFile,
        nullptr, // Необязательная структура KEVENT
        nullptr, nullptr, // Без APC
        &ioStatus,
        buffer,
        (ULONG)min((LONGLONG)size, fileSize.QuadPart), // Количество байтов
        &offset,  // Смещение
        nullptr); // Необязательный ключ
    if (!NT_SUCCESS(status))
        break;

    bytes = (ULONG)ioStatus.Information;

    // Запись в целевой файл
    status = ZwWriteFile(
        hTargetFile,      // Целевой дескриптор
```

```
        nullptr,          // Необязательная структура KEVENT
        nullptr, nullptr, // APC-вызов, контекст APC
        &ioStatus,        // Статус ввода/вывода
        buffer,           // Данные для записи
        bytes,            // Количество записываемых байтов
        &writeOffset,     // Смещение
        nullptr);         // Необязательный ключ

    if (!NT_SUCCESS(status))
        break;

    // Обновление счетчика байтов и смещений
    offset.QuadPart += bytes;
    writeOffset.QuadPart += bytes;
    fileSize.QuadPart -= bytes;
}
```

Цикл продолжается, пока остаются байты для передачи. Мы начинаем с размера файла и уменьшаем его для каждого переданного фрагмента. Вся фактическая работа выполняется функциями `ZwReadFile` и `ZwWriteFile`. Операция чтения возвращает количество фактически переданных байтов в поле `Information` структуры `IO_STATUS_BLOCK`, которое используется для инициализации локальной переменной `bytes`, используемой при операции записи.

Остается сделать последний шаг. Поскольку операция может перезаписать предыдущую резервную копию (которая может быть больше текущей), необходимо 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 свой аргумент в итоговых версиях.

Напоследок необходимо освободить все занятые ресурсы:

```
    if (buffer)
        ExFreePool(buffer);
    if (hSourceFile)
        FltClose(hSourceFile);
    if (hTargetFile)
        FltClose(hTargetFile);

    return status;
}
```

Обратный вызов после освобождения

Для чего нужен еще один обратный вызов? Наш контекст присоединен к файлу, что означает, что он будет удален только при удалении файла, чего может и не быть. Контекст необходимо освободить при закрытии файла клиентом.

Есть две операции, которые могут пригодиться в такой ситуации: `IRP_MJ_CLOSE` и `IRP_MJ_CLEANUP`. Операция закрытия (первая) интуитивно кажется более подходящей, потому что она должна вызываться при закрытии последнего дескриптора для файла. Тем не менее из-за кэширования это иногда происходит недостаточно быстро. Лучше обрабатывать операцию `IRP_MJ_CLEANUP`, которая фактически означает, что объект файла более не нужен, даже если последний дескриптор еще не закрыт. Этот момент хорошо подходит для освобождения контекста (если он существует).

Обратный вызов после освобождения похож на любой другой обратный вызов:

```
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);
```

Необходимо получить контекст файла, и если он существует, освободить все динамически выделенные ресурсы непосредственно перед их удалением:

```
    FileContext* context;

    auto status = FltGetFileContext(FltObjects->Instance,
        FltObjects->FileObject, (PFLT_CONTEXT*)&context);
    if (!NT_SUCCESS(status) || context == nullptr) {
        // Контекста нет, продолжить нормальное выполнение
        return FLT_POSTOP_FINISHED_PROCESSING;
    }

    if (context->FileName.Buffer)
        ExFreePool(context->FileName.Buffer);
    FltReleaseContext(context);
    FltDeleteContext(context);
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

Тестирование драйвера

Чтобы протестировать драйвер, можно установить его в системе, как это обычно делается, а затем попытаться работать с файлами в каталоге documents или pictures.

В следующем примере я создал в папке documents файл hello.txt с текстом «Hello, world!», сохранил файл, затем изменил содержимое на «Goodbye, world!» и сохранил файл снова. На рис. 10.11 показана программа NtfsStreams с этим файлом. В следующем окне командной строки выводится текущее содержимое файла:

```
c:\users\pavel\documents>type hello.txt
Goodbye, world!
```

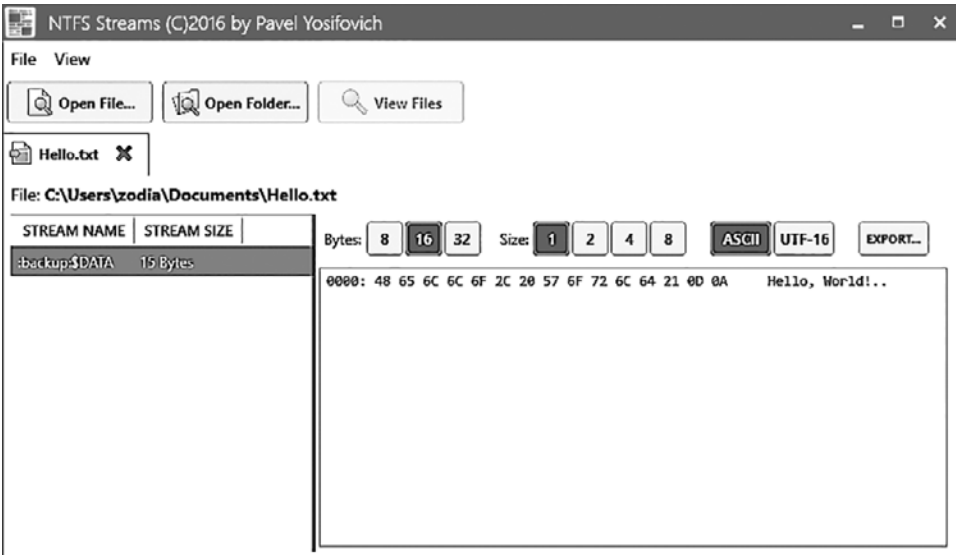


Рис. 10.11. Программа NtfsStreams с резервной копией файла

Восстановление из резервных копий

Как восстановить содержимое резервной копии? Необходимо скопировать содержимое потока `:backup` поверх «нормального» содержимого файла. К сожалению, функция API `CopyFile` этого сделать не сможет, так как она не может получать альтернативные потоки данных.

Напишем программу для решения этой задачи.

Создадим новый проект консольного приложения с именем `FileRestore`. Добавьте следующие директивы `#include` в файл pch.h:

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

Функция `main` должна получать имя файла в аргументе командной строки:

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

Программа открывает два файла: один указывает на поток данных `:backup`, а другой — на «обычный» файл. Данные копируются по фрагментам по аналогии с кодом драйвера BackupFile — но в пользовательском режиме (весь код обработки ошибок опущен для краткости):

```
    // Генерирование полного имени потока данных
    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;
}
```

Обработка ошибок присутствует в полном исходном коде программы.

Расширьте драйвер, чтобы в файле сохранялся дополнительный поток данных с временем и датой создания резервной копии.

Взаимодействие с пользовательским режимом

В предыдущих главах был представлен один из способов взаимодействия между драйвером и клиентом пользовательского режима: использование `DeviceIoControl`. Безусловно, это хорошее решение, которое хорошо работает во многих сценариях. Впрочем, у него есть и недостатки: взаимодействие должно инициироваться клиентом пользовательского режима. Если у драйвера имеются данные, которые он хотел бы отправить клиенту (или клиентам) пользовательского режима, он не сможет сделать это напрямую. Он должен сохранить эти данные и ожидать запроса на получение данных со стороны клиента.

Диспетчер фильтров предоставляет альтернативный механизм двустороннего взаимодействия между мини-фильтром файловой системы и клиентами пользовательского режима, позволяющий любой стороне отправить информацию другой стороне и даже ожидать ответа.

Мини-фильтр создает порт передачи данных с фильтром вызовом `FltCreateCommunicationPort`, чтобы зарегистрировать обратные вызовы для взаимодействия с клиентами и передачи сообщений. Клиент пользовательского режима подключается к порту вызовом `FilterConnectCommunicationPort`, получая дескриптор порта.

Мини-фильтр отправляет сообщение клиенту(-ам) пользовательского режима функцией `FltSendMessage`. И наоборот, клиент пользовательского режима вызывает функцию `FilterGetMessage`, чтобы ожидать поступления сообщения, или функцию `FilterSendMessage` для отправки сообщения драйверу. Если драйвер ожидает получить ответ, то клиент пользовательского режима вызывает `FilterReplyMessage` с ответом.

Создание порта передачи данных

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);
```

Описание параметров FltCreateCommunicationPort:

- Filter — непрозрачный указатель, возвращаемый функцией FltRegisterFilter.
- ServerPort — выходной непрозрачный дескриптор, который используется во внутренней реализации для прослушивания входных сообщений из пользовательского режима.
- ObjectAttributes — стандартная структура атрибутов, которая должна содержать имя порта на сервере и дескриптор безопасности, который разрешал бы клиентам пользовательского режима подключаться к порту (подробнее об этом позднее).
- ServerPortCookie — необязательный указатель, определяемый драйвером, по которому можно различать несколько открытых портов в обратных вызовах сообщения.
- ConnectNotifyCallback — обратный вызов, который должен быть предоставлен драйвером; вызывается при подключении нового клиента к порту.
- DisconnectNotifyCallback — вызывается при отключении клиента пользовательского режима от порта.
- MessageNotifyCallback — обратный вызов, активизируемый при поступлении сообщения к порту.
- MaxConnections — максимальное количество клиентов, которые могут подключаться к порту. Должно быть больше нуля.

Для успешного вызова FltCreateCommunicationPort драйвер должен подготовить атрибуты объекта и дескриптор безопасности. Простейший дескриптор безопасности может быть создан функцией FltBuildDefaultSecurityDescriptor:

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

После этого 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 входит в пространство имен диспетчера объектов и может просматриваться в программе WinObje после создания порта. Набор флагов должен включать `OBJ_KERNEL_HANDLE`, в противном случае вызов завершается неудачей. Обратите внимание: последний аргумент содержит дескриптор безопасности, определенный ранее. Теперь драйвер готов к вызову `FltCreateCommunicationPort`, что обычно происходит после того, как драйвер вызовет `FltRegisterFilter` (потому что возвращенный непрозрачный объект фильтра необходим для вызова), но до вызова `FltStartFiltering`, чтобы порт был готов в тот момент, когда начнется реальная фильтрация:

```
PFLT_PORT ServerPort;

status = FltCreateCommunicationPort(FilterHandle, &ServerPort, &portAttr,
nullptr,
    PortConnectNotify, PortDisconnectNotify, PortMessageNotify, 1);

// Освобождение дескриптора безопасности
FltFreeSecurityDescriptor(sd);
```

Подключение из пользовательского режима

Для подключения к открытому порту клиенты пользовательского режима вызывают функцию `FilterConnectCommunicationPort`, которая объявляется следующим образом:

```
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);
```

Краткая сводка параметров:

- `lpPortName` — имя порта (например, `\MyPort`). Обратите внимание: с дескриптором безопасности по умолчанию, созданным драйвером, подключаться смогут только процессы административного уровня.
- `dwOptions` — обычно равен 0, но содержит `FLT_PORT_FLAG_SYNC_HANDLE` в Windows 8.1 и выше — признак того, что возвращаемый дескриптор дол-

жен работать только синхронно. Неясно, для чего это нужно, потому что по умолчанию дескриптор все равно используется синхронно.

- `lpContext` и `wSizeOfContext` — поддерживают возможность передачи буфера драйверу во время подключения. Например, эта возможность может использоваться как средство аутентификации — драйверу передается некий пароль или маркер, а драйвер отказывает тем запросам на подключение, которые не соответствуют заранее определенному механизму аутентификации. Такой подход обычно рекомендуется применять в драйверах коммерческого уровня, чтобы неизвестные клиенты не могли «перехватить» порт обмена данными у нормальных клиентов.
- `lpSecurityAttributes` — структура пользовательского режима `SECURITY_ATTRIBUTES`, обычно содержит `NULL`.
- `hPort` — выходной дескриптор, используемый клиентом в дальнейшем для отправки и получения сообщений.

Функция активизирует обратный вызов уведомления клиента, а ее объявление выглядит так:

```
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` — уникальный дескриптор порта клиента; драйвер должен сохранить его и использовать каждый раз, когда ему потребуется взаимодействовать с клиентом. `ServerPortCookie` — то же значение, которое было задано драйвером в `FltCreateCommunicationPort`. Параметры `ConnectionContex` и `SizeOfContex` содержат необязательный буфер, отправленный клиентом. Наконец, `ConnectionPortCookie` содержит необязательное значение, которое драйвер может вернуть как признак данного клиента; это значение передается в клиентских функциях уведомления об отключении и поступлении сообщений.

Если драйвер согласен принять подключение клиента, он возвращает `STATUS_SUCCESS`. В противном случае клиент получит код ошибки `HRESULT` в `FilterConnectCommunicationPort`.

Если вызов `FilterConnectCommunicationPort` завершится успехом, клиент может начать обмен данными с драйвером, и наоборот.

Отправка и получение сообщений

Драйвер мини-фильтра может отправлять сообщения клиентам функцией `FltSendMessage`, которая объявляется 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);
```

Первые два параметра вам уже знакомы. Драйвер может отправить любой буфер, описанный `SenderBuffer`, с длиной `SenderBufferLength`. Обычно драйвер определяет некоторую структуру в общем заголовочном файле, который также может быть включен клиентом для правильной интерпретации полученного буфера.

Также возможно, что драйвер ожидает получить ответ; в таком случае параметр `ReplyBuffer` должен быть отличен от `NULL`, при этом максимальная длина ответа хранится в `ReplyLength`. Наконец, `Timeout` сообщает, как долго драйвер готов ждать, пока сообщение достигнет клиента (а также ждать возможного ответа). Тайм-аут имеет обычный формат, который я повторю для удобства:

- Если указатель равен `NULL`, драйвер готов ожидать неограниченное время.
- Если значение положительно, оно содержит абсолютное время в 100-наносекундных единицах от полуночи 1 января 1601 года.
- Если значение отрицательно, оно содержит относительное время (наиболее распространенный случай) в тех же 100-наносекундных единицах. Например, для задания односекундного интервала следует указать значение `-100000000`. Чтобы задать интервал в x миллисекунд, умножьте значение x на `-10000`.

Драйвер не должен возвращать `NULL` из обратного вызова, потому что это означает, что клиент в настоящее время не ведет прослушивания и поток блокируется до того, как он начнет прослушивать, — чего может и не произойти. Лучше задать некоторое ограниченное значение. А еще лучше, если запрос не нужен немедленно: можно воспользоваться рабочим элементом для отправки сообщения и ожидать в течение более длительного времени в случае необходимости (за дополнительной информацией о рабочих элементах обращайтесь к главе 6, хотя диспетчер фильтров имеет собственный API рабочих элементов).

С точки зрения клиента он может ожидать сообщения от драйвера с помощью функции `FilterGetMessage`, которой передается дескриптор порта, полученный при подключении, буфер и размер для входного сообщения, а также структура `OVERLAPPED`, которая может использоваться для выполнения вызова в асинхронном режиме (без блокирования). Полученный буфер всегда содержит заголовок типа `FILTER_MESSAGE_HEADER`, за которым следуют фактические данные, отправленные драйвером. Структура `FILTER_MESSAGE_HEADER` определяется следующим образом:

```
typedef struct _FILTER_MESSAGE_HEADER {
    ULONG ReplyLength;
    ULONGLONG MessageId;
} FILTER_MESSAGE_HEADER, *PFILTER_MESSAGE_HEADER;
```

Если ожидается ответ, `ReplyLength` указывает максимальное количество ожидаемых байтов. Поле `MessageId` содержит идентификатор, по которому различаются сообщения; оно должно использоваться клиентом при вызове `FilterReplyMessage`.

Клиент может сам инициировать сообщение функцией `FilterSendMessage`, что в конечном итоге приводит к активизации обратного вызова драйвера, зарегистрированного в `FltCreateCommunicationPort`. Функция `FilterSendMessage` может указать буфер с отправляемым сообщением и необязательный буфер для ответа, который может быть возвращен мини-фильтром.

> За полной информацией обращайтесь к документации `FilterSendMessage` и `FilterReplyMessage`.

Расширенный драйвер File Backup

Доработаем драйвер File Backup, чтобы он отправлял уведомления клиенту пользовательского режима при создании резервной копии файла. Начнем с определения глобальной переменной для хранения состояния, связанного с портом передачи данных:

```
PFLT_PORT FilterPort;
PFLT_PORT SendClientPort;
```

`FilterPort` — серверный порт драйвера, а `SendClientPort` — клиентский порт после подключения (мы будем поддерживать только один клиент).

Функцию `DriverEntry` необходимо изменить, чтобы она создавала порт обмена данными, как описано в предыдущем разделе. Код после успешного выполнения `FltRegisterFilter` (обработка ошибок опущена):

```
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);

// Непосредственное начало фильтрации ввода/вывода

status = FltStartFiltering(gFilterHandle);
```

Драйвер разрешает подключение к порту только одному клиенту (последняя 1 в вызове `FltCreateCommunicationPort`), как это обычно бывает, когда мини-фильтр работает на пару со службой пользовательского режима.

Обратный вызов `PortConnectNotify` активизируется при попытке подключения со стороны клиента. Наш драйвер просто сохраняет порт клиента и возвращает признак успеха:

```
_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;
}
```

При отключении клиента активизируется вызов `PortDisconnectNotify`. Важно закрыть в этот момент клиентский порт, в противном случае 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 — сообщения отправляются только драйвером, поэтому обратный вызов `PostMessageNotify` имеет пустую реализацию.

Теперь необходимо отправить сообщение при успешном резервном копировании файла. Для этого мы определим структуру сообщения, общую для драйвера и клиента, в отдельном заголовочном файле FileBackupCommon.h:

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

Сообщение содержит длину имени файла и само имя файла. Сообщение не имеет фиксированного размера и зависит от длины имени файла. В обратном вызове перед записью, после успешного резервного копирования файла, необходимо 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 клиент, и если подключен, выделяем буфер подходящего размера для хранения имени файла и копируем его в буфер (функцией `RtlCopyMemory`, аналогичной `memcpy`).

Теперь все готово к отправке сообщения с фиксированным тайм-аутом:

```
    LARGE_INTEGER timeout;
    timeout.QuadPart = -10000 * 100; // 100 мс
    FltSendMessage(gFilterHandle, &SendClientPort, msg, len,
        nullptr, nullptr, &timeout);
    ExFreePool(msg);
}
```

Наконец, в функции выгрузки фильтра необходимо закрыть порт фильтра (перед вызовом `FltUnregisterFilter`):

```
FltCloseCommunicationPort(FilterPort);
```

Клиент пользовательского режима

Построим просто клиент, который открывает порт и прослушивает сообщения о копируемых файлах. Создадим простое консольное приложение с именем FileBackupMon. Добавим в файл pch.h следующие директивы `#include`:

```
#include <Windows.h>
#include <fltUser.h>
```

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

fltuser.h — заголовок пользовательского режима, в котором объявляются функции `Filter`*Xxx* (они не входят в windows.h). В cpp-файл необходимо добавить библиотеку, в которой реализованы эти функции:

```
#pragma comment(lib, "fltlib")
```

Также библиотеку можно добавить в свойствах проекта в разделе Linker (категория Input). Поместить ее в исходный файл проще и надежнее, так как изменения в свойствах проекта не повлияют на настройку. Без этой библиотеки будут появляться ошибки компоновщика «Неразрешенная внешняя ссылка».

Функция `main` должна открыть порт обмена данными:

```
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 можно выделить буфер для входных сообщений и в бесконечном цикле ожидать сообщений.

После того как сообщение будет получено, оно отправляется для обработки:

```
BYTE buffer[1 << 12]; // 4 Кб
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 выделяется статически, потому что сообщение состоит в основном из имени файла, так что 4-килобайтного буфера более чем достаточно. После того как сообщение будет получено, мы передаем тело сообщения вспомогательной функции `HandleMessage` (не забыв пропустить заголовок).

После этого остается сделать с данными что-то полезное:

```
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());
}
```

Строка создается по указателю и длине (к счастью, в стандартном классе C++ wstring присутствует этот удобный конструктор). Это важно, потому что строка не завершается NULL-символом (хотя мы могли бы заполнить буфер нулями перед получением каждого сообщения, чтобы строка гарантированно завершалась нулями).

Чтобы порт был открыт успешно, клиентское приложение должно выполняться с повышенными привилегиями.

Отладка

Отладка мини-фильтра файловой системы принципиально не отличается от отладки любого другого драйвера ядра. Однако в пакет средств отладки для Windows входит специальная DLL-библиотека расширения fltkd.dll со специализированными командами, упрощающими отладку мини-фильтров. Эта DLL-библиотека не входит в число библиотек расширения, загружаемых по умолчанию, поэтому команды должны использоваться с «полным именем», включающим префикс fltkd и команду. Также DLL-библиотеку можно загрузить явно командой .load, после чего команды можно использовать напрямую.

В табл. 10.3 приведены некоторые команды из расширения fltkd с краткими описаниями.

Таблица 10.3. Команды отладчика fltkd.dll

Команда	Описание
!help	Выводит список команд с краткими описаниями
!filters	Выводит информацию обо всех загруженных мини-фильтрах
!filter	Выводит информацию для заданного адреса фильтра
!instance	Выводит информацию для заданного адреса экземпляра
!volumes	Выводит все объекты томов
!volume	Выводит подробную информацию о заданном адресе тома
!portlist	Выводит список серверных портов для заданного фильтра
!port	Выводит информацию для заданного клиентского порта

Пример сеанса, в котором используются некоторые из перечисленных 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 вместо удаления перемещает файлы в корзину.
2. Расширьте драйвер File Backup возможностью выбора каталогов, в которых будут создаваться резервные копии.
3. Расширьте драйвер File Backup возможностью включения нескольких резервных копий, с ограничением по некоторому принципу, например размеру файла, дате или максимальному количеству резервных копий.
4. Измените драйвер File Backup, чтобы в резервной копии сохранялись только измененные данные вместо всего файла.
5. Предложите собственные идеи для драйвера мини-фильтра файловой системы!

Итоги

Эта глава была посвящена мини-фильтрам файловой системы — мощным драйверам, способным перехватывать любые действия в файловой системе. Мини-фильтры — обширная тема, и эта глава поможет вам сделать первые шаги на этом интересном и непростом пути. Дополнительную информацию вы найдете в документации WDK, примерах WDK на Github и в некоторых блогах.

В следующей (последней) главе будут рассмотрены различные методы разработки драйверов и другие общие темы, которые не имеют прямого отношения ни к одной из предшествующих глав.

Глава 11
Разное

В последней главе книги рассматриваются различные темы, которые не имеют прямого отношения ни к одной из предшествующих глав.

В этой главе:

- Цифровые подписи драйверов
- Driver Verifier
- Использование платформенного API
- Драйверы-фильтры
- Device Monitor
- Перехват операций драйверами
- Библиотеки режима ядра

Цифровые подписи драйверов

Драйверы режима ядра — единственный официальный механизм выполнения кода в режиме ядра Windows. А это означает, что драйверы ядра могут привести к сбою системы или другой форме нестабильности системы. В ядре Windows нет различий между «более важными» и «менее важными» драйверами. Естественно, компания Microsoft хотела бы, чтобы система Windows работала стабильно, без сбоев или неполадок. Начиная с Windows Vista, в 64-разрядных системах компания Microsoft требует, чтобы драйверы снабжались цифровой подписью — специальным сертификатом, выданным центром сертификации. Без цифровой подписи драйвер не загрузится.

Гарантирует ли наличие подписи качество драйвера? Гарантирует ли наличие подписи, что в системе не будет сбоев? Нет. Оно гарантирует лишь то, что фай-

лы драйвера не изменились после их публикации, а сам публикатор проверен. Цифровая подпись — не панацея от ошибок драйверов, но она дает некоторую уверенность в качестве драйвера.

Компания Microsoft требует, чтобы драйверы оборудования проходили тесты WHQL (Windows Hardware Quality Lab) — систему жестких проверок стабильности и функциональности драйверов. Если драйвер проходит эти тесты, он получает печать качества Microsoft, которую издатель драйвера может предъявить как признак качества и доверия драйвера. Кроме того, после прохождения тестов WHQL драйвер становится доступным в системе Windows Update, что важно для некоторых издателей.

Начиная с Windows 10 версии 1607 («Anniversary update»), для установленных «с нуля» систем (не обновления более ранних версий) с включенным механизмом Secure Boot компания Microsoft требует, чтобы драйверы были подписаны как Microsoft, так и издателем. Это относится ко всем типам драйверов, не только к драйверам оборудования. Microsoft предоставляет веб-портал, на который издатель может отправить свой драйвер (уже снабженный подписью издателя). Microsoft тестирует полученный драйвер, в итоге снабжает его подписью и возвращает издателю. Возможно, Microsoft потребуется какое-то время на то, чтобы вернуть подписанный драйвер при первой загрузке, но позднее итерации заметно ускоряются (до нескольких часов).

> Для выдачи цифровой подписи достаточно отправить только двоичные файлы драйвера (исходный код не нужен).

На рис. 11.1 показан файл образа драйвера от Nvidia с цифровыми подписями от Nvidia и Microsoft в системе Windows 10 19H1.

Первым шагом для создания цифровой подписи драйвера должно стать получение соответствующего сертификата от центра сертификации (например, Verisign, Globalsign, Digicert, Symantec и др). — по крайней мере для кода режима ядра. Центр сертификации подтверждает подлинность компании, и если все проходит нормально — выдает сертификат. Загруженный сертификат может быть установлен в хранилище сертификатов на машине. Так как сертификат должен храниться в секрете без утечек, обычно он устанавливается на специализированной машине, на которой происходит сборка, а процесс выдачи цифровой подписи драйвера становится частью процесса сборки.

Фактическая операция назначения цифровой подписи осуществляется программой SignTool.exe — частью Windows SDK. Вы можете использовать Visual Studio для подписывания драйвера, если сертификат установлен в хранилище сертификатов на локальной машине. На рис. 11.2 показаны свойства цифровых подписей в Visual Studio.

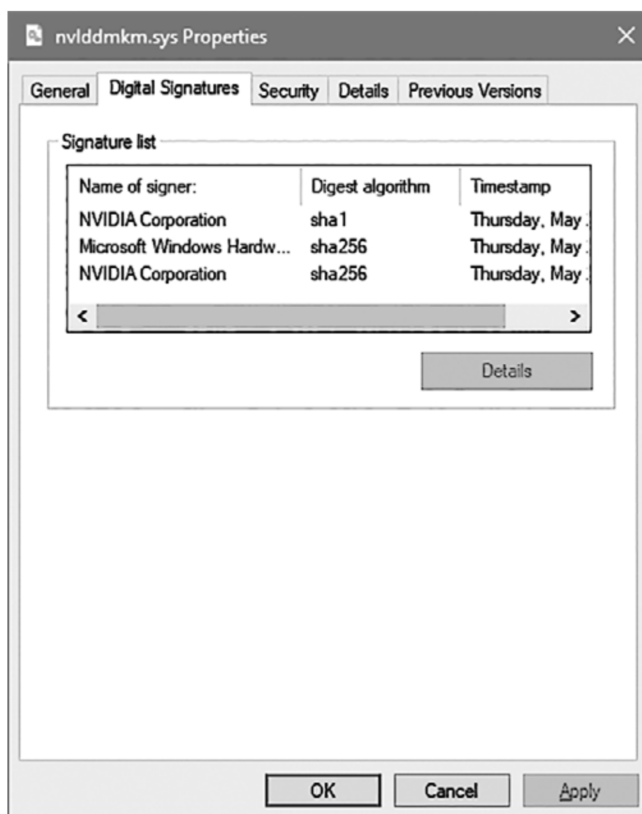


Рис. 11.1. Драйвер с подписями издателя и Microsoft

Visual Studio поддерживает два типа подписей: тестовые и серийные. В системе тестовых подписей обычно используется тестовый сертификат (локально сгенерированный сертификат, не пользующийся глобальным доверием). Тестовые подписи позволяют протестировать драйвер в системах, настроенных для использования тестовых подписей, как это делалось в книге. При использовании серийных подписей драйвер подписывается реальным сертификатом для коммерческого использования.

Чтобы сгенерировать тестовый сертификат в Visual Studio, выберите тип сертификата, как показано на рис. 11.3.

На рис. 11.4 показан пример серийной подписи для конечной сборки драйвера в Visual Studio. Обратите внимание: для подписи следует использовать алгоритм дайджеста SHA256 вместо старого и менее надежного алгоритма SHA1.

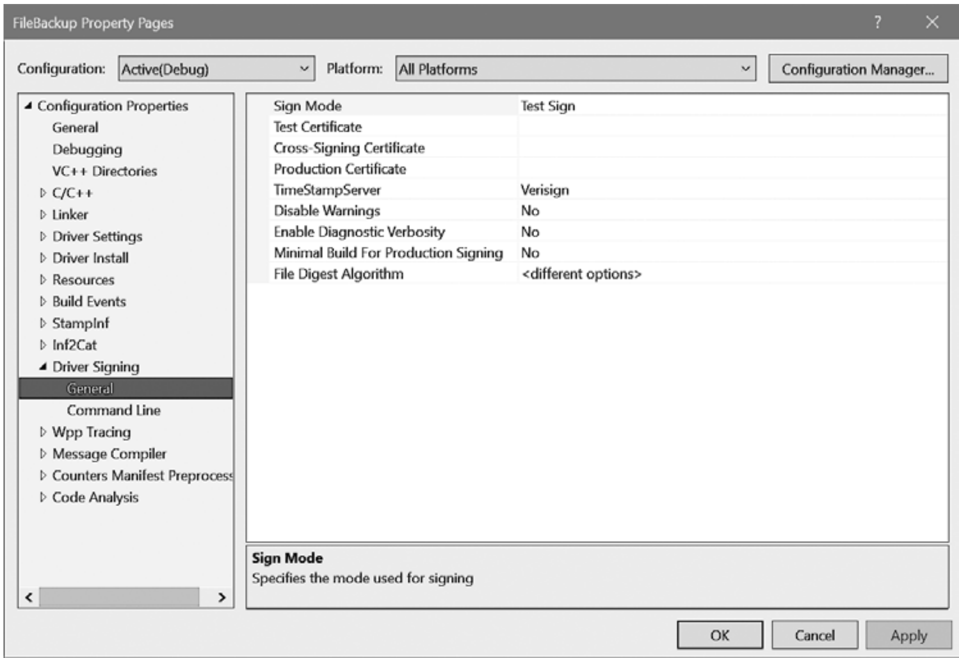


Рис. 11.2. Страница подписи драйверов в Visual Studio

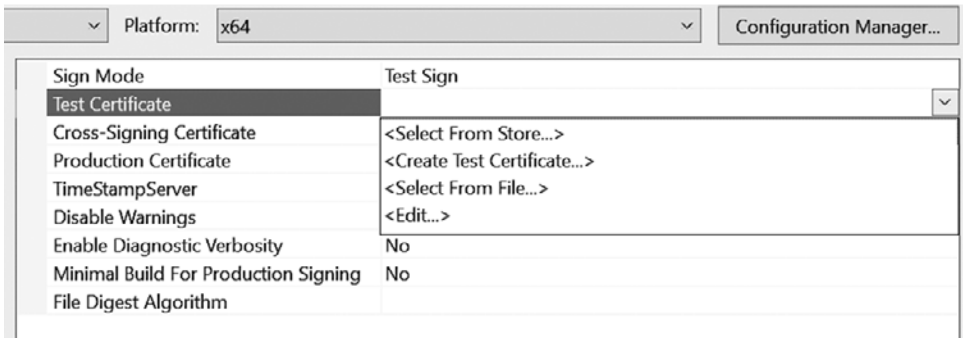


Рис. 11.3. Выбор типа сертификата в Visual Studio

Различные процедуры регистрации и назначения подписей драйверам выходит за рамки книги. В последнее время ситуация усложнилась из-за введения новых правил и процедур Microsoft.

За подробностями обращайтесь к официальной документации[1].

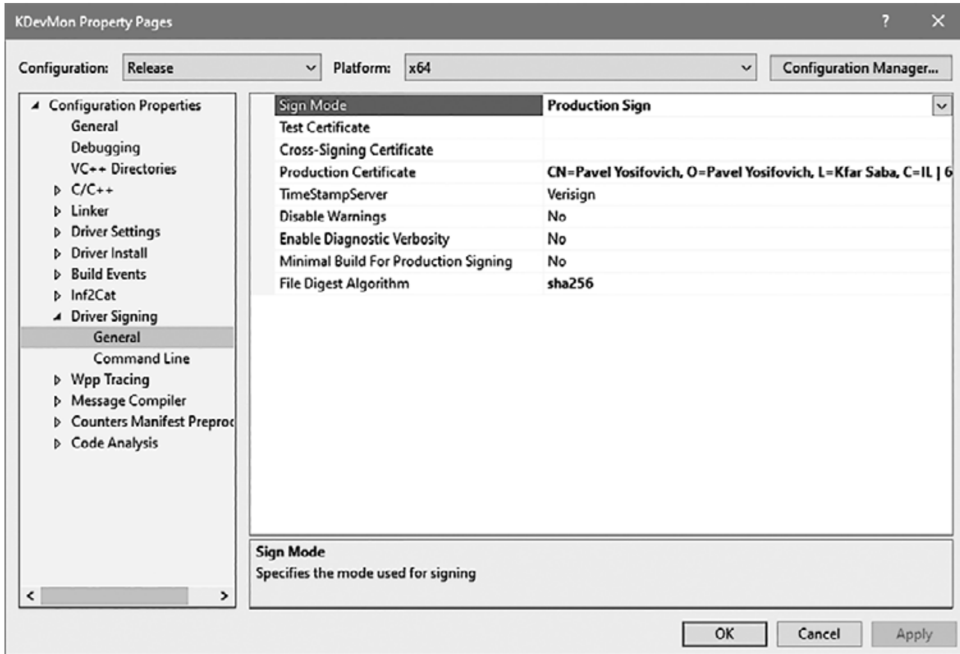


Рис. 11.4. Серийная подпись драйвера в Visual Studio

Driver Verifier

Driver Verifier — встроенная программа, существующая в Windows еще со времен Windows 2000. Она предназначена для выявления ошибок драйверов и плохих практик программирования. Допустим, ваш драйвер по какой-то причине вызывает «синий экран смерти» (BSOD), но код драйвера не присутствует ни в одном стеке вызовов в файле аварийного дампа. Обычно это означает, что ваш драйвер сделал что-то такое, что не было фатально в тот момент (например, запись с выходом за границу одного из выделенных буферов), но, к сожалению, эта память была выделена другому драйверу или ядру. В этот момент никакого сбоя нет. Тем не менее в какой-то момент драйвер или ядро пытается использовать эти данные, записанные за границей буфера, что, скорее всего, приведет к фатальному сбою системы. Связать этот сбой с драйвером-нарушителем не-

[1] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-.

просто. Driver Verifier предлагает возможность выделить память для драйвера в своем «специальном» пуле, в котором страницы с более высокими и низкими адресами недоступны, поэтому любой выход за границу буфера приведет к немедленному фатальному сбою, что упростит выявление драйвера, вызвавшего проблемы.

Driver Verifier поддерживает графический интерфейс и интерфейс командной строки и может работать с любыми драйверами — исходный код не нужен. Чтобы начать работу с Verifier, проще всего ввести команду `verifier` в диалоговом окне Выполнить или провести поиск `verifier` при открытии меню Пуск. В любом случае Verifier отображает исходный интерфейс, показанный на рис. 11.5.

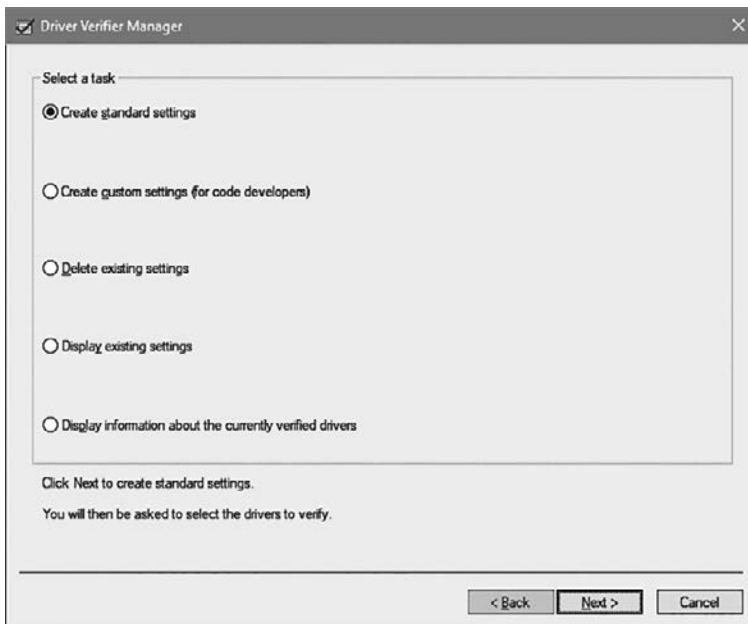


Рис. 11.5. Исходное окно Driver Verifier

Здесь необходимо выбрать два параметра: тип проверок, которые должны выполняться драйвером, и драйверы, которые должны проверяться. Первая страница мастера посвящена проверкам. Параметры, доступные на этой странице:

- Create standard settings — выбирает заранее определенный набор выполняемых проверок. Полный список доступных проверок приведен на второй странице, каждая проверка помечена флагом Standard или Additional. Этот параметр автоматически выбирает все проверки с флагом Standard.

- Create custom settings — позволяет уточнить выбор проверок с перечислением всех доступных проверок, показанных на рис. 11.6.
- Delete existing settings — удаляет все существующие настройки Verifier.
- Display existing settings — выводит текущие настройки и драйверы, к которым применяются эти проверки.
- Display information about the currently verified drivers — выводит собранную информацию для драйверов, выполняемых под управлением Verifier в предыдущем сеансе.

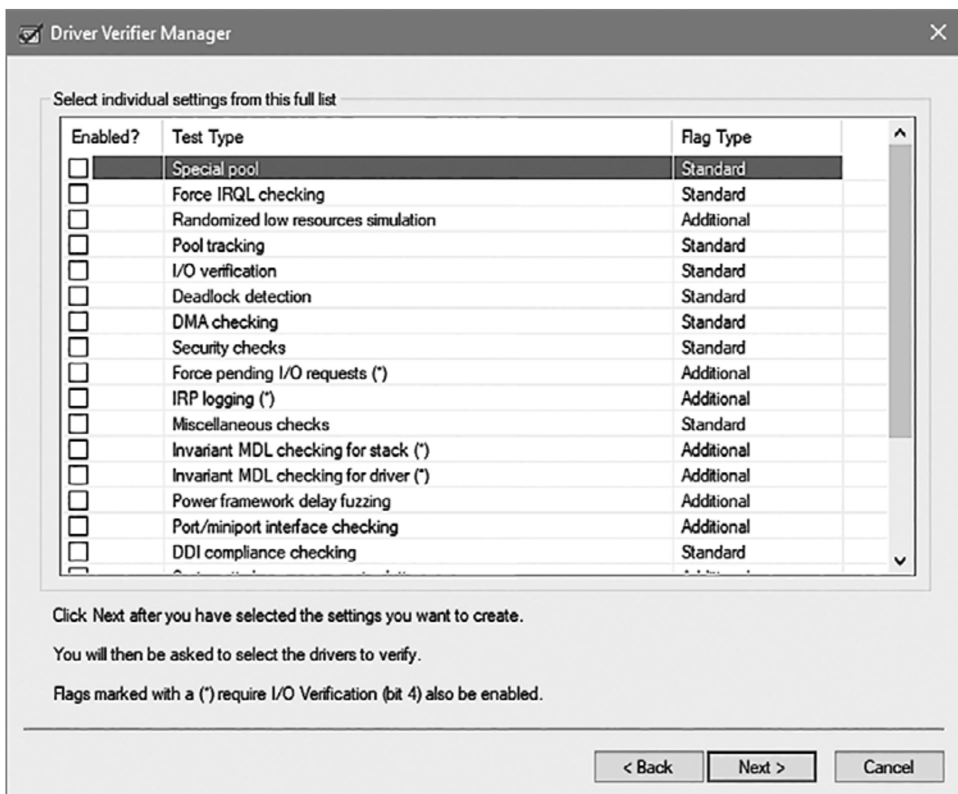


Рис. 11.6. Выбор настроек Driver Verifier

При выборе режима Create custom settings выводится список настроек Verifier, заметно расширившийся за время существования Driver Verifier. Флаг Standard означает, что настройка является частью стандартных настроек, которые выбираются на первой странице мастера. После того как настройки будут выбраны,

Verifier отображает следующее окно для выбора драйверов для выполнения с этими настройками (рис. 11.7).

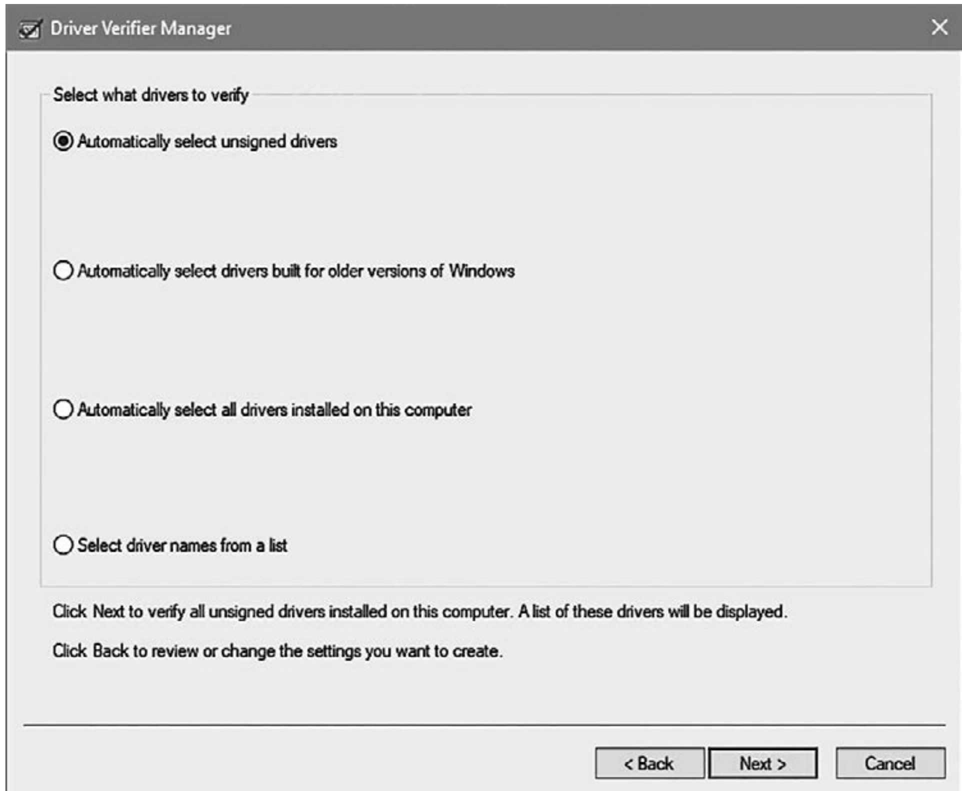


Рис. 11.7. Исходный выбор драйвера в Driver Verifier

Возможные варианты:

- Automatically select unsigned drivers — автоматический выбор неподписанных драйверов. Этот вариант актуален прежде всего для 32-разрядных систем, так как 64-разрядные системы должны иметь подписанные драйверы (кроме режима тестовых подписей). Кнопка Next выводит список таких драйверов (в большинстве систем список будет пустым).
- Automatically select drivers built for older versions of Windows — унаследованная настройка для драйверов оборудования NT 4. Для современных систем интереса в основном не представляет.
- Automatically select all drivers installed on the computer — автоматический выбор всех драйверов, установленных на компьютере. Теоретически может быть

полезен, если вы столкнулись с системой, в которой происходят сбои, но никто не может определить, из-за какого драйвера они происходят. Тем не менее использовать эту настройку не рекомендуется, так как она замедляет работу машины (выполнение Verifier не бесплатно), потому что Verifier перехватывает различные операции (на основании своих настроек) и обычно требует дополнительных затрат памяти. Таким образом, в таком сценарии лучше выбрать первые (допустим) 15 драйверов и посмотреть, обнаруживает ли Verifier некорректно работающий драйвер; если нет — выбрать следующие 15 драйверов, и т. д.

- Select driver names from a list — выбор имен драйверов из списка. Лучший вариант: Verifier выводит список драйверов, выполняемых в системе в настоящий момент (рис. 11.8). Если драйвер не выполняется, кнопка Add currently not loaded driver(s) позволяет перейти к соответствующему SYS-файлу(-ам).

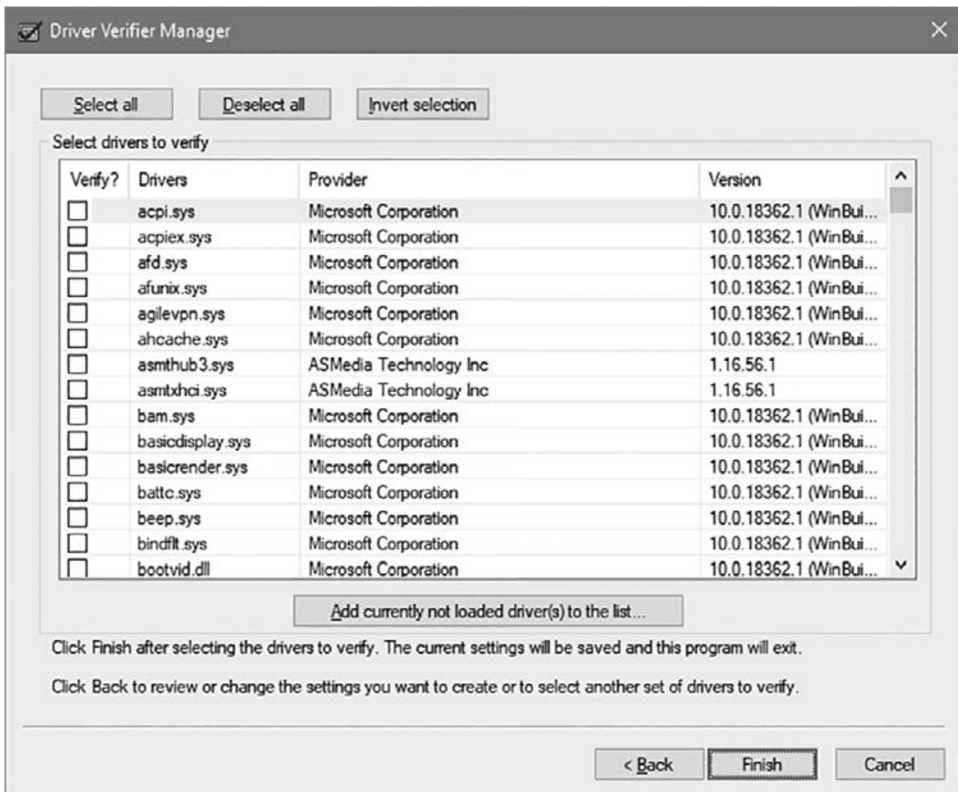


Рис. 11.8. Выбор конкретного драйвера в Driver Verifier

Наконец, кнопка Finish закрепляет настройки до момента отмены. Обычно систему приходится перезапустить, чтобы программа Verifier могла инициализировать себя и организовать перехват обращений к драйверам, особенно если они работают в настоящее время.

Пример сеанса Driver Verifier

Начнем с простого примера, в котором используется программа NotMyFault из Sysinternals. Как упоминалось в главе 6, с помощью этой программы можно инициировать различные сбои в системе. На рис. 11.9 показан основной пользовательский интерфейс программы NotMyFault. В некоторых вариантах фатальный сбой системы происходит немедленно, при этом драйвер MyFault.sys присутствует в стеке вызовов потока-инициатора. Такие сбои диагностируются достаточно легко. С другой стороны, в режиме Buffer overflow немедленный фатальный сбой системы не гарантирован. Если сбой в системе произойдет позднее, вряд ли MyFault.sys будет присутствовать в стеке вызовов.

Проследите за тем, чтобы программа NotMyFault64.exe запускалась в 64-разрядной системе.

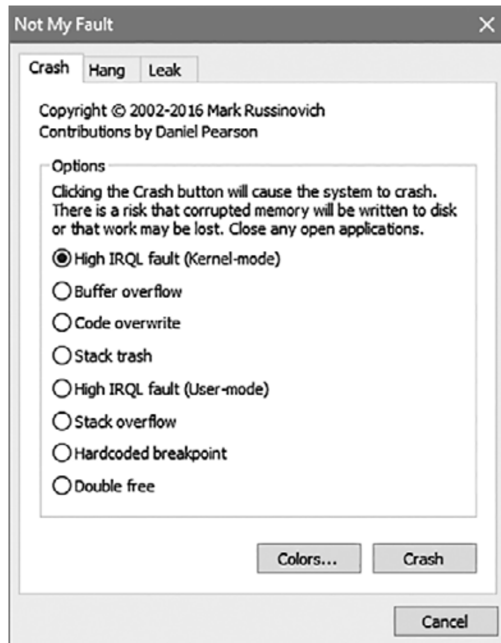


Рис. 11.9. Основной пользовательский интерфейс NotMyFault

Попробуем выполнить этот пример (на виртуальной машине). Возможно, для фактического возникновения сбоя системы придется несколько раз щелкнуть на кнопке Crash. На рис. 11.10 показан результат на виртуальной машине Windows 7 после нескольких щелчков на кнопке Crash и по прошествии нескольких секунд. Обратите внимание на код BSOD (`BAD_POOL_HEADER`). Уместно предположить, что сбой произошел из-за того, что при переполнении буфера были перезаписаны некие метаданные, относящиеся к выделению памяти из пула.

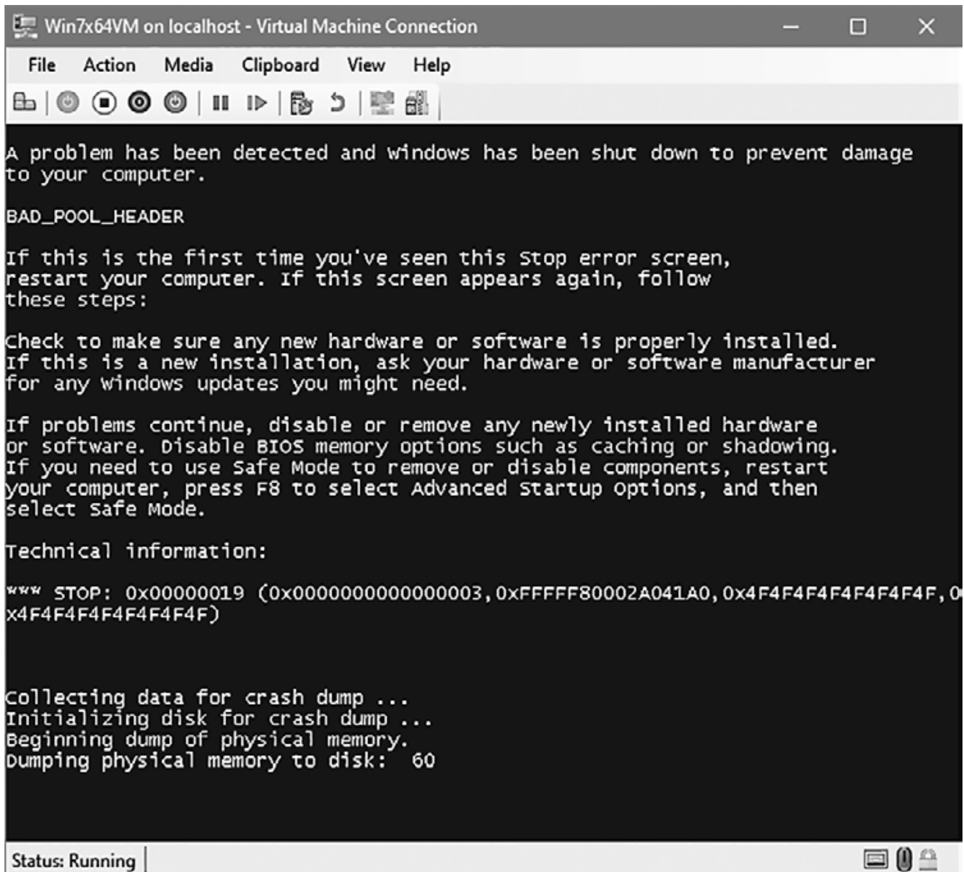


Рис. 11.10. Программа NotMyFault вызывает BSOD в Windows 7 переполнением буфера

При загрузке полученного файла дампа и просмотре стека вызовов будет получен следующий результат:

```
1: kd> k
#  Child-SP          RetAddr           Call Site
00 fffff880`054be828 fffff800`029e4263 nt!KeBugCheckEx
01 fffff880`054be830 fffff800`02bd969f nt!ExFreePoolWithTag+0x1023
02 fffff880`054be920 fffff800`02b0669b nt!ObpAllocateObject+0x12f
03 fffff880`054be990 fffff800`02c2f012 nt!ObCreateObject+0xdb
04 fffff880`054bea00 fffff800`02b1a7b2 nt!PspAllocateThread+0x1b2
05 fffff880`054bec20 fffff800`02b20d95 nt!PspCreateThread+0x1d2
06 fffff880`054beea0 fffff800`028aaad3 nt!NtCreateThreadEx+0x25d
07 fffff880`054bf5f0 fffff800`028a02b0 nt!KiSystemServiceCopyEnd+0x13
08 fffff880`054bf7f8 fffff800`02b29a60 nt!KiServiceLinkage
09 fffff880`054bf800 fffff800`0286ac1a nt!RtlpCreateUserThreadEx+0x138
0a fffff880`054bf920 fffff800`0285c1c0 nt!ExpWorkerFactoryCreateThread+0x92
0b fffff880`054bf9e0 fffff800`02857dd0 nt!ExpWorkerFactoryCheckCreate+0x180
0c fffff880`054bfa60 fffff800`028aaad3 nt!NtReleaseWorkerFactoryWorker+0x1a0
0d fffff880`054bfae0 00000000`76e1ac3a nt!KiSystemServiceCopyEnd+0x13
```

Очевидно, MyFault.sys нигде не встречается. Кстати говоря, команда `analyze -v` тоже умом не блещет: она считает, что виновником является модуль `nt`.

Теперь попробуем провести тот же эксперимент с Driver Verifier. Выберите стандартные настройки, перейдите в каталог System32\Drivers и найдите файл MyFault.sys (если он не выполняется в настоящий момент). Перезагрузите систему, снова запустите NotMyFault, выберите вариант Buffer overflow и щелкните на кнопке Crash. Как нетрудно заметить, сбой происходит немедленно с выдачей экрана BSOD вроде показанного на рис. 11.11.

Сам экран BSOD уже содержит достаточно полную информацию. Файл дампа подтверждает предварительные выводы следующим стеком вызовов:

```
0: kd> k
#  Child-SP          RetAddr           Call Site
00 fffff880`0651c378 fffff800`029ba462 nt!KeBugCheckEx
01 fffff880`0651c380 fffff800`028ecb96 nt!MmAccessFault+0x2322
02 fffff880`0651c4d0 fffff880`045f1c07 nt!KiPageFault+0x356
03 fffff880`0651c660 fffff880`045f1f88 myfault+0x1c07
04 fffff880`0651c7b0 fffff800`02d63d56 myfault+0x1f88
05 fffff880`0651c7f0 fffff800`02b43c7a nt!IovCallDriver+0x566
06 fffff880`0651c850 fffff800`02d06eb1 nt!IopSynchronousServiceTail+0xfa
07 fffff880`0651c8c0 fffff800`02b98296 nt!IopXxxControlFile+0xc51
08 fffff880`0651ca00 fffff800`028eead3 nt!NtDeviceIoControlFile+0x56
09 fffff880`0651ca70 00000000`777e98fa nt!KiSystemServiceCopyEnd+0x13
```

У нас нет данных символических имен для MyFault.sys, но очевидно, именно он является виновником.

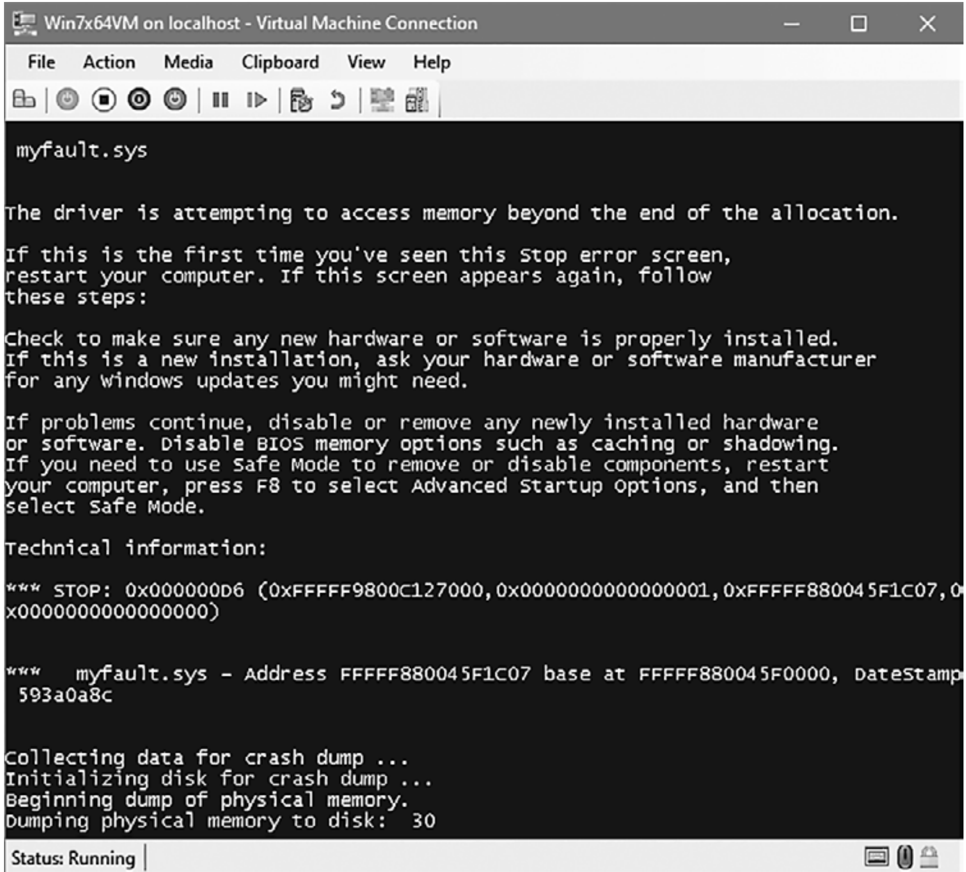


Рис. 11.11. Программа NotMyFault вызывает BSOD в Windows 7 переполнением буфера при активной программе Verifier

Другой пример: применим стандартные настройки Verifier для драйвера DelProtect3 из главы 10. После того как драйвер будет установлен, попробуем добавить папку для защиты от удаления:

```
delprotectconfig3 add c:\temp
```

Происходит системный фатальный сбой, инициированный Verifier, с кодом `BAD_POOL_CALLER` (0xc2). Открыв файл дампа и выполнив команду `!analyze -v`, вы получите данные анализа, часть которых приведена ниже:

```
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at
                                                 a bad IRQL le\
```

```
vel or double freeing the same allocation, etc.
Arguments:
Arg1: 000000000000009b, Attempt to allocate pool with a tag of zero. This would \
make the pool untrackable and worse, corrupt the existing tag tables.
Arg2: 0000000000000001, Pool type
Arg3: 000000000000000c, Size of allocation in bytes
Arg4: fffff8012dcd1297, Caller's address.

FAULTING_SOURCE_CODE:
   113:   return pUnicodeString;
   114: }
   115:
   116: wchar_t* kstring::Allocate(size_t chars, const wchar_t* src) {
>  117:   auto str = static_cast<wchar_t*>(ExAllocatePoolWithTag(m_Pool,
sizeof(WCHAR) * (chars + 1), m_Tag));
   118:   if (!str) {
   119:       KdPrint(("Failed to allocate kstring of length %d chars\n", \
chars));
   120:       return nullptr;
   121:   }
   122:  if (src) {
```

В самом деле, объект `kstring`, использованный в коде, не указал ненулевой тег для своих выделений памяти. Код-нарушитель является частью функции `ConvertDosNameToNtName`:

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

Проблема решается достаточно просто:

```
kstring symLink(L"\\??\\", PagedPool, DRIVER_TAG);
```

Возможно, класс `kstring` следует изменить так, чтобы он требовал явной передачи тега, вместо того чтобы использовать нуль по умолчанию.

Использование платформенного API

Как было показано в главах 1 и 10, платформенный API системы Windows, предоставляемый пользовательскому режиму через библиотеку NtDll.Dll, является шлюзом к функциональности ядра. Платформенный API использует косвенный вызов реальных функций исполнительной системы, для чего номер системной сервисной функции помещается в регистр процессора (EAX для Intel/AMD), а специальная машинная команда (`syscall` или `sysenter` на платформах Intel/AMD) инициирует переход из пользовательского режима в режим ядра, а также активизацию диспетчера системных функций, использующего значение этого регистра для вызова фактической системной функции.

Драйверы ядра могут пользоваться тем же API, как уже было показано на примере различных Zw-функций. Тем не менее документировано лишь малое количество таких функций, особенно функции, связанные с получением информации (NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread и другие аналогичные функции). Вам уже встречалась функция NtQueryInformationProcess, показанная ниже в Zw-версии для удобства обсуждения:

```
NTSTATUS ZwQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_ PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

Перечисление PROCESSINFOCLASS огромно — для сборки 18362 в WDK для него приведено около 70 значений. Внимательно просмотрев список, вы увидите, что некоторые значения в нем отсутствуют. В официальной документации описано всего 6 значений (на момент написания), что весьма прискорбно. Более того, реально поддерживаемый список намного длиннее того, который официально предоставляет компания Microsoft. Доступно большое количество интересной информации, которая может использоваться драйвером при необходимости.

К счастью, проект с открытым кодом на Github, называемый Process Hacker, предоставляет большую часть отсутствующей информации в виде платформенных определений функций API, перечислений и структур, с которыми должны работать эти функции.

Process Hacker — программа с открытым ходом, сходная с Process Explorer. В ней реализованы некоторые возможности, отсутствующие в Process Explorer (на момент написания книги). Репозиторий находится по адресу https://github.com/processhacker/processhacker.

Process Hacker содержит все определения платформенного API в виде отдельного проекта, удобного для использования в других проектах (https://github.com/processhacker/phnt). Просто для наглядного сравнения: PROCESSINFOCLASS в этом репозитории в настоящее время содержит 99 записей (присутствуют все значения) вместе со структурами данных, на которые рассчитаны различные значения из перечисления.

Единственное, о чем следует предупредить: теоретически компания Microsoft может в любой момент изменить почти любое из этих определений, так как большинство из них не документировано. Тем не менее это крайне маловероятно, так как изменение может нарушить работу многих приложений,

в том числе и приложений Microsoft. В частности, некоторые из этих «недокументированных» функций используются в Process Explorer. Тем не менее желательно тестировать приложения и драйверы, использующие эти функции, для всех версий Windows, в которых должны работать эти приложения или драйверы.

Драйверы-фильтры

Как было показано в главе 7, в модели драйверов Windows центральное место занимают *устройства*. Устройства могут располагаться друг над другом, образуя иерархию, так что устройство верхнего уровня первым получает доступ к поступившему запросу IRP. Данная модель может использоваться для драйверов файловой системы, которые использовались в главе 10, с помощью диспетчера фильтров, специализирующегося на фильтрах файловой системы. Тем не менее модель файловой системы имеет общий характер и может применяться к устройствам других типов. В этом разделе будет более подробно рассмотрена общая модель фильтрации устройств, которая может быть применена к широкому диапазону устройств. Одни из этих устройств могут быть связаны с физическими устройствами, другие — нет.

API режима ядра предоставляет ряд функций для размещения устройств в вертикальной иерархии. Вероятно, простейшей является функция `IoAttachDevice`, которая получает объект присоединяемого устройства и объект целевого именованного устройства, к которому оно присоединяется. Прототип выглядит так:

```
NTSTATUS IoAttachDevice (
    PDEVICE_OBJECT SourceDevice,
    _In_ PUNICODE_STRING TargetDevice,
    _Out_ PDEVICE_OBJECT *AttachedDevice);
```

Результатом работы функции (помимо статуса) является другой объект устройства, к которому был фактически присоединен объект `SourceDevice`. Он необходим из-за того, что присоединение к именованному устройству, не находящемуся на вершине своего стека устройств, завершается успешно, но исходное устройство фактически присоединяется к верхнему устройству, которым может быть другой фильтр. Однако важно получить реальное устройство, к которому присоединено само исходное устройство, потому что это устройство должно стать целью запросов, если драйвер пожелает распространять их вниз по стеку устройств (рис. 11.12).

К сожалению, присоединение к объекту устройства требует большего объема работы. Как упоминалось в главе 7, устройство может приказать диспетчеру ввода/вывода помочь с обращением к буферу устройства с буферизованным

вводом/выводом или прямым вводом/выводом (для запросов IRP_MJ_READ и IRP_MJ_WRITE), устанавливая соответствующие флаги в поле Flags структуры DEVICE_OBJECT. В иерархическом сценарии задействовано несколько устройств; какое же устройство должно определять, как диспетчер ввода/вывода должен помогать при работе с буферами ввода/вывода? Оказывается, это всегда верхнее устройство в стеке. Следовательно, наше новое устройство-фильтр должно скопировать значение флагов DO_BUFFERED_IO и DO_DIRECT_IO с устройства, поверх которого оно фактически располагается. По умолчанию у устройства, только что созданного вызовом IoCreateDevice, ни один из этих флагов не устанавливается, и если новое устройство не скопирует эти биты, с большой вероятностью это приведет к некорректной работе и даже фатальному сбою целевого устройства, так как оно не ожидает, что выбранный им метод буферизации не будет соблюдаться.

Есть несколько других настроек, которые необходимо скопировать с присоединенного устройства, чтобы новый фильтр выглядел так же с точки зрения системы ввода/вывода. Эти настройки будут рассмотрены позднее, когда мы займемся построением полного примера фильтра.

Какое имя устройства должно передаваться IoAttachDevice? Это должен быть объект именованного устройства из пространства имен диспетчера объектов, которое можно просмотреть в инструментарии WinObj, уже использовавшемся нами ранее. Многие объекты именованных устройств находятся в каталоге \Device\, но некоторые расположены в других местах. Например, если вы захотите присоединить объект устройства-фильтра к объекту устройства Process Explorer, будет использоваться имя \Device\ProcExp152 (регистр символов в имени не учитывается).

Также для присоединения к другому объекту устройства используются функции IoAttachDeviceToDeviceStack и IoAttachDeviceToDeviceStackSafe. Обе

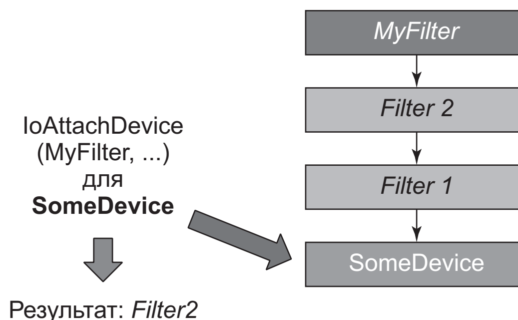


Рис. 11.12. Присоединение к именованному устройству

функции вместо имени устройства получают другой объект устройства для присоединения. Эти функции удобны прежде всего при построении фильтров, зарегистрированных для драйверов физических устройств, когда объект целевого устройства предоставляется как часть узла устройства (который также упоминался в главе 7). Обе функции возвращают фактический объект устройства, как и `IoAttachDevice`. В случае неудачи функция `Safe` возвращает признак `NTSTATUS`, тогда как первая функция возвращает `NULL`. В остальном эти функции идентичны.

Как правило, код режима ядра может получить указатель на объект именованного устройства вызовом функции `IoGetDeviceObjectPointer`, которая возвращает объект устройства и объект файла, открытый для этого устройства на основании имени устройства. Прототип функции выглядит так:

```
NTSTATUS IoGetDeviceObjectPointer (
    _In_ PUNICODE_STRING ObjectName,
    _In_ ACCESS_MASK DesiredAccess,
    _Out_ PFILE_OBJECT *FileObject,
    _Out_ PDEVICE_OBJECT *DeviceObject);
```

Параметр `DesiredAccess` содержит требуемый уровень доступа — обычно `FILE_READ_DATA` или любое другое значение, соответствующее объектам файлов. Счетчик ссылок для возвращаемого объекта файла увеличивается, так что драйвер должен в конечном итоге уменьшить значение счетчика (`ObDereferenceObject`), чтобы предотвратить утечку объекта файла. Возвращаемый объект устройства может использоваться в аргументе `IoAttachDeviceToDeviceStack(Safe)`.

Реализация драйвера-фильтра

Драйвер-фильтр должен присоединить объект устройства к целевому устройству, которому требуется фильтрация. Позднее мы поговорим о том, когда должно происходить такое присоединение, а пока будем считать, что в какой-то момент вызывается одна из функций присоединения. Так как новый объект устройства становится верхним устройством в стеке устройств, любой запрос, не поддерживаемый драйвером, возвращается клиенту с ошибкой «Операция не поддерживается». Это означает, что функция `DriverEntry` фильтра должна регистрироваться для всех первичных кодов функций, если она хочет гарантировать, что объект устройства продолжит нормально работать. Например, настройка может выглядеть так:

```
for (int i = 0; i < ARRAYSIZE(DriverObject->MajorFunction); i++)
    DriverObject->MajorFunction[i] = HandleFilterFunction;
```

Этот фрагмент связывает все коды первичных функций с одной функцией. Функция HandleFilterFunction должна как минимум вызвать драйвер более низкого уровня с использованием объекта устройства, полученного от одной из функций присоединения. Конечно, поскольку драйвер является фильтром, он захочет выполнить дополнительную или другую работу для запросов, которые представляют для него интерес, а все запросы, которые его не интересуют, должны быть переданы устройству более низкого уровня, иначе это устройство будет работать некорректно.

Операция «перенаправить и забыть» очень часто применяется в фильтрах. Давайте посмотрим, как реализуется эта функциональность. Фактическая передача IRP другому устройству осуществляется вызовом IoCallDriver. Однако перед вызовом этой функции текущий драйвер должен подготовить следующую позицию стека ввода/вывода, которая должна использоваться драйвером более низкого уровня. Напомню, что изначально диспетчер ввода/вывода инициализирует только первую позицию стека ввода/вывода. Задача каждого уровня — инициализировать следующую позицию стека ввода/вывода перед вызовом IoCallDriver для передачи запроса IRP вниз по стеку устройств.

Драйвер может вызвать функцию IoGetNextIrpStackLocation для получения указателя на структуру IO_STACK_LOCATION следующего уровня и инициализировать ее. Однако в большинстве случаев драйвер хочет всего лишь передать нижнему уровню ту информацию, которую получил сам. В этом может помочь функция IoCopyCurrentIrpStackLocationToNext, смысл которой понятен без объяснений. Тем не менее эта функция **не ограничивается** простым копированием позиции стека ввода/вывода по следующему принципу:

```
auto current = IoGetCurrentIrpStackLocation(Irp);
auto next = IoCopyCurrentIrpStackLocationToNext(Irp);
*next = *current;
```

Почему? Причина не столь очевидна и имеет отношение к функции завершения. Вспомните, о чем говорилось в главе 7: драйвер может определить функцию завершения, которая будет уведомляться при завершении IRP драйвером более низкого уровня (IoSetCompletionRoutine/Ex). Указатель на функцию завершения (и определяемый драйвером аргумент контекста) хранится в позиции стека ввода/вывода; таким образом, простое копирование продублирует функцию завершения более верхнего уровня (если она определена), а мы вряд ли этого хотим. Именно этого избегает функция IoCopyCurrentIrpStackLocationToNext.

Однако существует и более удачное решение, если драйверу не нужна функция завершения и он хочет просто «перенаправить и забыть» без затрат на копирование данных в позиции стека ввода/вывода. Для этого позиция стека ввода/

вывода пропускается так, чтобы следующий драйвер более низкого уровня увидел ту же позицию стека ввода/вывода.

```
IoSkipCurrentIrpStackLocation(Irp);
status = IoCallDriver(LowerDeviceObject, Irp);
```

Функция `IoSkipCurrentIrpStackLocation` просто уменьшает внутренний указатель на позицию стека ввода/вывода в IRP, а функция `IoCallDriver` увеличивает ее, в результате чего драйвер более низкого уровня видит ту же позицию стека ввода/вывода, что и текущий уровень, без какого-либо копирования; этот способ распространения IRP является предпочтительным, если драйвер не хочет вносить изменения в запрос и обходится без функции завершения.

Присоединение фильтров

Когда драйвер вызывает функции присоединения? В идеале — при создании нижележащего устройства (цели присоединения), то есть при построении узла устройства. Такая ситуация часто встречается в фильтрах драйверов физических устройств, когда фильтры могут регистрироваться в именованных параметрах `UpperFilters` и `LowerFilters`, упоминавшихся в главе 7. Для этих фильтров фактическое создание нового объекта устройства и присоединение его к существующим стекам драйверов выполняется в обратном вызове, назначенном в поле `AddDevice`, доступ к которому осуществляется через объект драйвера:

```
DriverObject->DriverExtension->AddDevice = FilterAddDevice;
```

Обратный вызов `AddDevice` активизируется в тот момент, когда система Plug & Play идентифицирует новое физическое устройство, принадлежащее драйверу. Прототип функции выглядит так:

```
NTSTATUS AddDeviceRoutine (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PDEVICE_OBJECT PhysicalDeviceObject);
```

Система ввода/вывода предоставляет драйверу объект устройства, находящийся на нижнем уровне стека (`PhysicalDeviceObject` или PDO), который должен использоваться при вызове `IoAttachDeviceToDeviceStack(Safe)`. PDO — одна из причин, по которым `DriverEntry` не подходит для вызова функции присоединения — в этот момент объект PDO еще не был получен. Кроме того, в системе может появиться второе устройство того же типа (например, вторая USB-камера), и в этом случае функция `DriverEntry` вообще не будет вызвана — только функция `AddDevice`.

Пример реализации функции `AddDevice` для драйвера-фильтра (обработка ошибок опущена):

```
struct DeviceExtension {
    PDEVICE_OBJECT LowerDeviceObject;
};

NTSTATUS FilterAddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT PDO) {
    PDEVICE_OBJECT DeviceObject;
    auto status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,
        FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
    status = IoAttachDeviceToDeviceStackSafe(
        DeviceObject,             // Присоединяемое устройство
        PDO,                      // Целевое устройство
        &ext->LowerDeviceObject); // Фактический объект устройства

    // Скопировать информацию из присоединенного устройства

    DeviceObject->DeviceType = ext->LowerDeviceObject->DeviceType;

    DeviceObject->Flags |= ext->LowerDeviceObject->Flags &
        (DO_BUFFERED_IO | DO_DIRECT_IO);

    // Важно для физических устройств

    DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
    DeviceObject->Flags |= DO_POWER_PAGABLE;

    return status;
}
```

Несколько важных замечаний по поводу этого кода.

- Объект устройства создается без имени. Целевое устройство является именованным, а IRP-запрос обращен именно к нему, поэтому предоставлять собственное имя не нужно. Фильтр будет активизирован в любом случае.
- В вызове `IoCreateDevice` во втором аргументе указывается ненулевой размер, который приказывает диспетчеру ввода/вывода выделить дополнительный буфер (`DeviceExtension`) наряду со структурой `DEVICE_OBJECT`. До настоящего момента для управления состоянием устройства использовались глобальные переменные, потому что объект устройства был только один. Тем не менее драйвер фильтра может создать несколько объектов устройств и присоединиться к разным стекам устройств, что усложнит связывание объектов устройств с состоянием. Механизм расширения устройств позволяет легко получить устройствос учетом самого объекта устройства. В приведенном выше коде в качестве состояния сохраняется объект устройства более низ-

кого уровня, но при необходимости структуру можно расширить для включения дополнительной информации.

- Информация частично копируется из объекта устройства более низкого уровня, так что система ввода/вывода воспринимает наш фильтр как само целевое устройство. Говоря конкретнее, мы копируем тип устройства и флаги метода буферизации.
- Наконец, мы исключаем флаг `DO_DEVICE_INITIALIZING` (изначально устанавливаемый системой ввода/вывода), чтобы сообщить диспетчеру Plug & Play, что устройство готово к работе. Флаг `DO_POWER_PAGABLE` сообщает, что IRP-запросы питания должны поступать на уровне IRQL < `DISPATCH_LEVEL`; фактически он является обязательным.

Для приведенного выше кода следующая реализация по принципу «перенаправить и забыть» использует устройство более низкого уровня, как описано в предыдущем разделе:

```
NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(ext->LowerDeviceObject, Irp);
}
```

Присоединение фильтров в произвольное время

В предыдущем разделе рассматривалось присоединение устройства-фильтра в функции обратного вызова `AddDevice`, которая вызывается диспетчером Plug & Play в момент построения узла устройства. Для драйверов, не связанных с оборудованием и не имеющих настроек реестра для фильтров, обратный вызов `AddDevice` не активизируется.

Для таких более общих случаев драйвер-фильтр теоретически может присоединять устройства-фильтры в любой момент времени, создавая объект устройства (`IoCreateDevice`) и затем используя одну из функций присоединения. Это означает, что целевое устройство уже существует, что оно уже работает, и в какой-то момент ему назначается фильтр. Драйвер должен позаботиться о том, чтобы это небольшое «прерывание» не имело отрицательных последствий для целевого устройства.

Многие операции, представленные в предыдущих разделах, актуальны и в этом случае — например, копирование флагов из устройства более низкого уровня.

При этом нужно предпринять дополнительные меры к тому, чтобы операции целевого устройства не были прерваны.

При помощи функции IoAttachDevice следующий код создает объект устройства и присоединяет его к другому объекту именованного устройства (обработка ошибок опущена):

```
// Фиксированное имя используется для простоты
UNICODE_STRING targetName = RTL_CONSTANT_STRING(L"\\Device\\SomeDeviceName");

PDEVICE_OBJECT DeviceObject;
auto status = IoCreateDevice(DriverObject, 0, nullptr,
    FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

PDEVICE_OBJECT LowerDeviceObject;
status = IoAttachDevice(DeviceObject, &targetName, &LowerDeviceObject);

// Копирование информации

DeviceObject->Flags |= LowerDeviceObject->Flags & (DO_BUFFERED_IO | \
DO_DIRECT_IO);

DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
DeviceObject->Flags |= DO_POWER_PAGABLE;
DeviceObject->DeviceType = LowerDeviceObject->DeviceType;
```

Внимательный читатель может заметить, что в этом коде присутствует неявная ситуация гонки. Сможете ли вы найти ее?

По сути, это тот же код, который использовался в обратном вызове AddDevice из предыдущего раздела. Тем не менее в том коде ситуации гонки не было. Это объяснялось тем, что целевое устройство еще не было активно — узел устройства строился, устройство за устройством, снизу вверх. Устройство еще не было способно получать запросы.

Сравните с приведенным кодом — целевое устройство работает и может быть занятым, когда внезапно появляется новый фильтр. Система ввода/вывода позаботится о том, чтобы при выполнении фактической операции присоединения не возникло проблем, но после того, как IoAttachDevice вернет управление (а на самом деле до этого), запросы продолжат поступать. Допустим, операция чтения поступает непосредственно после возвращения управления IoAttachDevice, но перед установкой флагов метода буферизации — диспетчер ввода/вывода увидит эти флаги как нулевые (режим не установлен), так как он проверяет только верхнее устройство, которым теперь стал наш новый фильтр! Таким образом, если целевое устройство использует прямой ввод/вывод (например), диспетчер ввода/вывода не заблокирует пользовательский буфер, не создаст MDL и т. д. Все это может привести к фатальному сбою системы, если целевой драйвер всегда предполагает, что значение Irp->MdlAddress (например) отлично от NULL.

> Окно возможности для такого сбоя очень мало, но лучше полностью исключить всякий риск.

Как устранить эту ситуацию гонки? Необходимо полностью подготовить новый объект устройства, перед тем как выполнять фактическое присоединение. Можно вызвать `IoGetDeviceObjectPointer` для получения объекта целевого устройства, скопировать нужную информацию в ваше устройство (на этот момент еще не присоединенное), и только после этого вызвать `IoAttachDeviceToDeviceStack(Safe)`. Полный пример будет приведен позднее в этой главе.

Напишите код для использования `IoGetDeviceObjectPointer` так, как описано выше.

Деинициализация фильтра

После того как фильтр будет присоединен, в какой-то момент он должен быть отсоединен. Эта операция выполняется вызовом `IoDetachDevice` с указателем объекта устройства более низкого уровня. Обратите внимание: в аргументе передается объект устройства более низкого уровня, а не объект устройства фильтра. Наконец, необходимо вызвать для объекта устройства фильтра функцию `IoDeleteDevice`, как это делалось в наших драйверах до настоящего момента.

Вопрос в том, когда должен вызываться этот код деинициализации. Если драйвер выгружается явно, то нормальная функция выгрузки должна выполнить завершающие операции. Тем не менее с фильтрами физических устройств возникают некоторые сложности. Иногда такие драйверы выгружаются из-за событий Plug & Play, например из-за отключения устройства от системы. Драйверы физических устройств получают запрос `IRP_MJ_PNP` с дополнительным кодом `IRP IRP_MN_REMOVE_DEVICE`, который указывает, что само устройство отсутствует, поэтому весь узел устройства должен быть уничтожен. Драйвер отвечает за правильную обработку этого запроса PnP, отсоединение от узла устройства и удаление устройства.

А следовательно, для драйверов физических устройств простой схемы «перенаправить и забыть» для `IRP_MJ_PNP` будет недостаточно. Для `IRP_MN_REMOVE_DEVICE` необходимо предусмотреть особую обработку. Пример кода:

```
NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
    auto ext = (DeviceExtension*)fido->DeviceExtension;
    auto stack = IoGetCurrentIrpStackLocation(Irp);

    UCHAR minor = stack->MinorFunction;
    IoSkipCurrentIrpStackLocation(Irp);
```

```
    auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
    if (minor == IRP_MN_REMOVE_DEVICE) {
        IoDetachDevice(LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    return status;
}
```

Подробнее о драйверах-фильтрах физических устройств

С фильтрами драйверов физических устройств возникают другие осложнения. В функции `FilterDispatchPnp` из предыдущего раздела возникла ситуация гонки. Проблема в том, что во время обработки некоторого IRP-запроса может поступить запрос на отключение устройства (обрабатываемый на другом процессоре, например). Это приведет к тому, что вызовы `IoDeleteDevice` в драйверах, которые являются частью узла устройства во время подготовки фильтра, будут отправлять другие запросы вниз по стеку устройств. Более подробное объяснение ситуации гонки выходит за рамки книги, однако нам все равно понадобится более совершенное решение.

Проблема решается с помощью объекта системы ввода/вывода, называемого *блокировкой удаления* и представленного структурой `IO_REMOVE_LOCK`. По сути, эта структура управляет счетчиком ссылок для количества незавершенных IRP-запросов, обрабатываемых в настоящее время, и событием, которое срабатывает при обнулении счетчика, если в настоящее время имеется незавершенная операция удаления устройства. Схема использования `IO_REMOVE_LOCK` выглядит примерно так:

1. Драйвер выделяет память для структуры в составе расширения устройства или в глобальной переменной и инициализирует ее однократным вызовом `IoInitializeRemoveLock`.
2. Для каждого запроса IRP драйвер захватывает блокировку удаления вызовом `IoAcquireRemoveLock`, перед тем как передать его устройству более низкого уровня. Если вызов завершается неудачей (`STATUS_DELETE_PENDING`), это означает, что имеется незавершенная операция удаления и драйвер должен немедленно вернуть управление.
3. После того как нижний драйвер завершит обработку IRP, блокировка удаления освобождается (`IoReleaseRemoveLock`).
4. При обработке `IRP_MN_REMOVE_DEVICE` перед отсоединением и удалением устройства вызывается функция `IoReleaseRemoveLockAndWait`. Вызов завершится успешно после того, как будет завершена обработка всех остальных IRP.

С учетом всего сказанного обобщенная схема диспетчеризации с передачей запросов вниз по иерархии должна быть изменена следующим образом (предполагается, что блокировка удаления уже была инициализирована):

```
struct DeviceExtension {
    IO_REMOVE_LOCK RemoveLock;
    PDEVICE_OBJECT LowerDeviceObject;
};

NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

    // Второй аргумент не используется в конечных сборках
    auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
    if(!NT_SUCCESS(status)) { // STATUS_DELETE_PENDING
        Irp->IoStatus.Status = status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
        return status;
    }
    IoSkipCurrentIrpStackLocation(Irp);
    status = IoCallDriver(ext->LowerDeviceObject, Irp);

    IoReleaseRemoveLock(&ext->RemoveLock, Irp);
    return status;
}
```

Обработчик `IRP_MJ_PNP` необходимо изменить для правильного использования блокировки удаления:

```
NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
    auto ext = (DeviceExtension*)fido->DeviceExtension;
    auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
    if(!NT_SUCCESS(status)) { // STATUS_DELETE_PENDING
        Irp->IoStatus.Status = status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
        return status;
    }

    auto stack = IoGetCurrentIrpStackLocation(Irp);
    UCHAR minor = stack->MinorFunction;

    IoSkipCurrentIrpStackLocation(Irp);
    auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
    if (minor == IRP_MN_REMOVE_DEVICE) {
        // Ожидать при необходимости
        IoReleaseRemoveLockAndWait(&ext->RemoveLock, Irp);

        IoDetachDevice(ext->LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    else {
        IoReleaseRemoveLock(&ext->RemoveLock, Irp);
    }
    return status;
}
```

Device Monitor

Вооружившись всей приведенной информацией, можно построить обобщенный драйвер, который может присоединяться к объектам устройств, как фильтры к другим драйверам. Это позволяет перехватывать запросы (почти) к любым устройствам, которые вас заинтересуют. Клиент пользовательского режима добавляет и удаляет устройства для фильтрации.

Создайте новый проект драйвера `Empty WDFM` с именем KDevMon, как это уже неоднократно делалось ранее. Драйвер должен быть способен присоединяться к разным устройствам, а также предоставлять собственный объект CDO (Control Device Object) для обработки конфигурационных запросов от клиента пользовательского режима. Объект CDO будет создаваться в `DriverEntry` как обычно, но управление присоединением будет осуществляться отдельно на основании запросов от клиента пользовательского режима.

Для управления устройствами, к которым применяется фильтрация, мы создадим вспомогательный класс с именем `DevMonManager`. Его основная задача — добавление и удаление устройств, к которым применяется фильтрация. Каждое устройство представляется следующей структурой:

```
struct MonitoredDevice {
    UNICODE_STRING DeviceName;
    PDEVICE_OBJECT DeviceObject;
    PDEVICE_OBJECT LowerDeviceObject;
};
```

Для каждого устройства необходимо хранить объект устройства-фильтра (создаваемого драйвером), объект более низкого уровня, к которому оно присоединяется, и имя устройства. Имя потребуется для отсоединения. Класс `DevMonManager` содержит массив структур `MonitoredDevice` фиксированного размера, быстрый мьютекс для защиты массива, а также некоторые вспомогательные функции. Основные компоненты `DevMonManager`:

```
const int MaxMonitoredDevices = 32;

class DevMonManager {
    public:
    void Init(PDRIVER_OBJECT DriverObject);
    NTSTATUS AddDevice(PCWSTR name);
    int FindDevice(PCWSTR name);
    bool RemoveDevice(PCWSTR name);
    void RemoveAllDevices();
    MonitoredDevice& GetDevice(int index);

    PDEVICE_OBJECT CDO;
private:
    bool RemoveDevice(int index);
```

```
private:
    MonitoredDevice Devices[MaxMonitoredDevices];
    int MonitoredDeviceCount;
    FastMutex Lock;
    PDRIVER_OBJECT DriverObject;
};
```

Добавление устройства для фильтрации

Наибольший интерес представляет функция `DevMonManager::AddDevice`, в которой выполняется присоединение. Разберем ее шаг за шагом.

```
NTSTATUS DevMonManager::AddDevice(PCWSTR name) {
```

Сначала необходимо захватить мьютекс на тот случай, если одновременно выполняется сразу несколько операций добавления/удаления/поиска. Затем выполняется ряд быстрых проверок, которые определяют, не заняты ли все элементы в массиве и не фильтруется ли уже устройство:

```
AutoLock locker(Lock);
if (MonitoredDeviceCount == MaxMonitoredDevices)
    return STATUS_TOO_MANY_NAMES;

if (FindDevice(name) >= 0)
    return STATUS_SUCCESS;
```

Далее ищется свободный индекс в массиве, по которому будет храниться информация о создаваемом фильтре:

```
for (int i = 0; i < MaxMonitoredDevices; i++) {
    if (Devices[i].DeviceObject == nullptr) {
```

Признаком свободного элемента является `NULL` в поле указателя на объект устройства в структуре `MonitoredDevice`. Затем функция пытается получить указатель на объект фильтруемого устройства вызовом `IoGetDeviceObjectPointer`:

```
UNICODE_STRING targetName;
RtlInitUnicodeString(&targetName, name);

PFILE_OBJECT FileObject;
PDEVICE_OBJECT LowerDeviceObject = nullptr;
auto status = IoGetDeviceObjectPointer(&targetName, FILE_READ_DATA,
    &FileObject, &LowerDeviceObject);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to get device object pointer (%ws) (0x%8X)\n", \
name, status));
    return status;
}
```

Результатом `IoGetDeviceObjectPointer` является верхний объект устройства, которым не обязательно является объект целевого устройства. И это нормально, потому что любая операция присоединения в действительности присоединя-

ется к вершине стека устройства. Конечно, вызов функции может завершиться неудачей — чаще всего это происходит из-за того, что устройство с указанным именем не существует.

Следующим шагом становится создание нового объекта устройства-фильтра и его инициализация, отчасти основанная на только что полученном указателе на объект устройства. В то же время необходимо заполнить структуру `MonitoredDevice` подходящими данными. Для каждого созданного устройства нужно иметь расширение устройства, в котором хранится объект устройства более низкого уровня, чтобы мы могли легко обратиться к нему в момент обработки IRP. Для этого определяется структура расширения устройства `DeviceExtension`, в которой может храниться этот указатель (в файле DevMonManager.h):

```
struct DeviceExtension {
    PDEVICE_OBJECT LowerDeviceObject;
};
```

Вернемся к `DevMonManager::AddDevice` — создадим объект устройства-фильтра:

```
PDEVICE_OBJECT DeviceObject = nullptr;
WCHAR* buffer = nullptr;

do {
    status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,
        FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status))
        break;
```

При вызове `IoCreateDevice` передается размер выделяемого расширения устройства, а также сама структура `DEVICE_OBJECT`. Расширение устройства сохраняется в поле `DeviceExtension` в `DEVICE_OBJECT`, поэтому оно всегда доступно в случае необходимости. На рис. 11.13 показан эффект вызова `IoCreateDevice`.

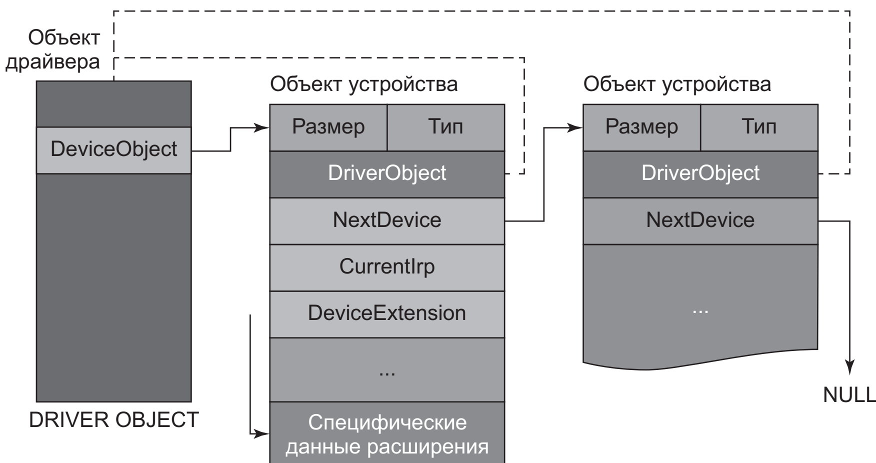


Рис. 11.13. Эффект вызова IoCreateDevice

Теперь можно продолжить инициализацию устройства и структуры `MonitoredDevice`:

```
// Выделение буфера для копирования имени устройства
buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool, targetName.Length, DRIVER_TAG);
if (!buffer) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}

auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

DeviceObject->Flags |= LowerDeviceObject->Flags & (DO_BUFFERED_IO | \
DO_DIRECT_IO);
DeviceObject->DeviceType = LowerDeviceObject->DeviceType;

Devices[i].DeviceName.Buffer = buffer;
Devices[i].DeviceName.MaximumLength = targetName.Length;
RtlCopyUnicodeString(&Devices[i].DeviceName, &targetName);
Devices[i].DeviceObject = DeviceObject;
```

> Строго говоря, мы могли использовать `LowerDeviceObject->DeviceType` вместо `FILE_DEVICE_UNKNOWN` при вызове `IoCreateDevice` и избавиться от хлопот с явным копированием поля `DeviceType`.

На этот момент новый объект устройства готов. Остается лишь присоединить его и завершить некоторые итоговые инициализации.

```
    status = IoAttachDeviceToDeviceStackSafe(
        DeviceObject,             // Объект устройства-фильтра
        LowerDeviceObject,        // Объект целевого устройства
        &ext->LowerDeviceObject); // Результат
    if (!NT_SUCCESS(status))
        break;

    Devices[i].LowerDeviceObject = ext->LowerDeviceObject;
    // Необходимо для драйверов оборудования
    DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
    DeviceObject->Flags |= DO_POWER_PAGABLE;
    MonitoredDeviceCount++;
} while (false);
```

Устройство присоединено, а полученный указатель сохраняется непосредственно в расширении устройства. Это важно, потому что сам процесс присоединения генерирует как минимум два IRP — `IRP_MJ_CREATE` и `IRP_MJ_CLEANUP`, и драйвер должен быть готов к их обработке. Как вы вскоре увидите, для такой обработки необходимо, чтобы объект устройства более низкого уровня был доступен в расширении устройства.

Остается лишь выполнить завершающие действия:

```
            if (!NT_SUCCESS(status)) {
                if (buffer)
                    ExFreePool(buffer);
                if (DeviceObject)
                    IoDeleteDevice(DeviceObject);
                Devices[i].DeviceObject = nullptr;
            }
            if (LowerDeviceObject) {
                // Освободить ссылку - объект не нужен
                ObDereferenceObject(FileObject);
            }
            return status;
        }
    }

    // никогда не должны попадать сюда
    NT_ASSERT(false);
    return STATUS_UNSUCCESSFUL;
}
```

Освобождение объекта файла — важный момент; ссылка была получена при вызове `IoGetDeviceObjectPointer`. Если не освободить ее, возникнет утечка ресурсов в режиме ядра. Следует заметить, что не обязательно (и более того, не следует) освобождать объект устройства, возвращенный при вызове `IoGetDeviceObjectPointer`, — ссылка на него будет автоматически освобождена при обнулении счетчика ссылок объекта файла.

Удаление устройства-фильтра

Удаление устройства из цепочки фильтрации происходит достаточно просто — нужно просто выполнить в обратном направлении то, что происходило в `AddDevice`:

```
bool DevMonManager::RemoveDevice(PCWSTR name) {
    AutoLock locker(Lock);
    int index = FindDevice(name);
    if (index < 0)
        return false;

    return RemoveDevice(index);
}

bool DevMonManager::RemoveDevice(int index) {
    auto& device = Devices[index];
    if (device.DeviceObject == nullptr)
        return false;

    ExFreePool(device.DeviceName.Buffer);
```

```
    IoDetachDevice(device.LowerDeviceObject);
    IoDeleteDevice(device.DeviceObject);
    device.DeviceObject = nullptr;

    MonitoredDeviceCount--;
    return true;
}
```

Важнейшие этапы — отсоединение устройства и его удаление. FindDevice — простая вспомогательная функция для поиска устройства по имени в массиве. Она возвращает индекс устройства в массиве или –1, если устройство не обнаружено:

```
int DevMonManager::FindDevice(PCWSTR name) {
    UNICODE_STRING uname;
    RtlInitUnicodeString(&uname, name);
    for (int i = 0; i < MaxMonitoredDevices; i++) {
        auto& device = Devices[i];
        if (device.DeviceObject &&
            RtlEqualUnicodeString(&device.DeviceName, &uname, TRUE)) {
            return i;
        }
    }
    return -1;
}
```

Единственный нюанс — проследить за тем, чтобы перед вызовом этой функции был захвачен быстрый мьютекс.

Инициализация и выгрузка

Функция DriverEntry выглядит довольно стандартно; она создает объект CDO, который может использоваться для добавления и удаления фильтров. Впрочем, существуют и некоторые различия. Но самое важное, что драйвер должен поддерживать все коды первичных функций, так как драйвер теперь служит двойной цели: с одной стороны, он предоставляет функциональность настройки для добавления и удаления устройств при вызове CDO, а с другой стороны, коды первичных функций будут использоваться клиентами самих фильтруемых устройств.

Работа над DriverEntry начнется с создания объекта CDO и получения доступа к нему по символической ссылке, как это уже неоднократно делалось ранее:

```
DevMonManager g_Data;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\KDevMon");
    PDEVICE_OBJECT DeviceObject;
```

```
    auto status = IoCreateDevice(DriverObject, 0, &devName,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
    if (!NT_SUCCESS(status))
        return status;

    UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\\??\\KDevMon");
    status = IoCreateSymbolicLink(&linkName, &devName);
    if (!NT_SUCCESS(status)) {
        IoDeleteDevice(DeviceObject);
        return status;
    }
    DriverObject->DriverUnload = DevMonUnload;
```

В этом коде нет ничего нового. Затем необходимо инициализировать все функции диспетчеризации, чтобы поддерживать все коды первичных функций:

```
for (auto& func : DriverObject->MajorFunction)
    func = HandleFilterFunction;

// эквивалентно:
// for (int i = 0; i < ARRAYSIZE(DriverObject->MajorFunction); i++)
//     DriverObject->MajorFunction[i] = HandleFilterFunction;
```

Аналогичный код уже встречался ранее в этой главе. В этом коде при помощи ссылки C++ все первичные функции связываются с функцией `HandleFilterFunction`, которая будет представлена ниже. Наконец, возвращенный объект для удобства сохраняется в глобальном объекте `g_Data` (`DevMonManager`) и инициализируется:

```
    g_Data.CDO = DeviceObject;
    g_Data.Init(DriverObject);

    return status;
}
```

Метод `Init` просто инициализирует быстрый мьютекс и сохраняет указатель на объект устройства для последующего использования функцией `IoCreateDevice` (описанной в предыдущем разделе).

> Мы не будем использовать блокировку удаления в этом драйвере для упрощения кода. Читатель может самостоятельно реализовать поддержку блокировки удаления, как было описано ранее в этой главе.

Прежде чем браться за обобщенную функцию диспетчеризации, стоит поближе познакомиться с функцией выгрузки. При выгрузке драйвера необходимо удалить символическую ссылку и CDO как обычно, но также необходимо отсоединиться от всех фильтров, активных в настоящий момент. Код выглядит так:

```
void DevMonUnload(PDRIVER_OBJECT DriverObject) {
```

```
    UNREFERENCED_PARAMETER(DriverObject);
    UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\\??\\KDevMon");
    IoDeleteSymbolicLink(&linkName);
    NT_ASSERT(g_Data.CDO);
    IoDeleteDevice(g_Data.CDO);

    g_Data.RemoveAllDevices();
}
```

Ключевым аспектом здесь является вызов `DevMonManager::RemoveAllDevices`. Функция устроена достаточно прямолинейно; вся основная работа выполняется вызовом `DevMonManager::RemoveDevice`:

```
void DevMonManager::RemoveAllDevices() {
    AutoLock locker(Lock);
    for (int i = 0; i < MaxMonitoredDevices; i++)
        RemoveDevice(i);
}
```

Обработка запросов

Функция диспетчеризации `HandleFilterFunction` — самая важная часть головоломки. Она будет вызываться для всех первичных функций, обращенных к одному из устройств-фильтров CDO. Функция должна каким-то образом отличать их; собственно, именно для этого мы ранее сохранили указатель на CDO. Наш объект CDO поддерживает функции создания, закрытия и `DeviceIoControl`. Исходный код выглядит так:

```
NTSTATUS HandleFilterFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    if (DeviceObject == g_Data.CDO) {
        switch (IoGetCurrentIrpStackLocation(Irp)->MajorFunction) {
            case IRP_MJ_CREATE:
            case IRP_MJ_CLOSE:
                return CompleteRequest(Irp);

            case IRP_MJ_DEVICE_CONTROL:
                return DevMonDeviceControl(DeviceObject, Irp);
        }
        return CompleteRequest(Irp, STATUS_INVALID_DEVICE_REQUEST);
    }
```

Если целевым устройством является наш объект CDO, то выбор осуществляется по самой первичной функции. Для создания и закрытия мы просто завершаем IRP успешно, вызывая вспомогательную функцию из главы 7:

```
NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS,
    ULONG_PTR information = 0);

NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR information) {
```

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

Для `IRP_MJ_DEVICE_CONTROL` вызывается функция `DevMonDeviceControl`, которая должна реализовать коды управляющих операций для добавления и удаления фильтров. Для всех остальных первичных функций IRP просто завершается с кодом ошибки «Операция не поддерживается».

Если объект устройства не является объектом CDO, то это должен быть один из наших фильтров. Здесь драйвер может сделать с запросом все, что посчитает нужным: зарегистрировать в журнале, проанализировать его, изменить и т. д. В нашем драйвере мы просто выведем в отладчике некоторую информацию о запросе, а затем направим его вниз устройству, находящемуся под фильтром.

Сначала получим расширение устройства, чтобы получить доступ к устройству более низкого уровня:

```
auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
```

Затем извлекается поток, выдавший запрос, — для этого мы обращаемся к данным IRP, а затем получаем идентификаторы потока и процесса вызывающей стороны:

```
auto thread = Irp->Tail.Overlay.Thread;
HANDLE tid = nullptr, pid = nullptr;
if (thread) {
    tid = PsGetThreadId(thread);
    pid = PsGetThreadProcessId(thread);
}
```

В большинстве случаев текущим потоком оказывается поток, который выдал исходный запрос, но это не обязательно — может оказаться, что фильтр более высокого уровня получил запрос, по какой-то причине не стал передавать его дальше немедленно, а позднее передал уже из другого потока.

Теперь можно вывести идентификаторы потока и процесса, а также тип запрашиваемой операции:

```
auto stack = IoGetCurrentIrpStackLocation(Irp);

DbgPrint("Intercepted driver: %wZ: PID: %d, TID: %d, MJ=%d (%s)\n",
    &ext->LowerDeviceObject->DriverObject->DriverName,
    HandleToUlong(pid), HandleToUlong(tid),
    stack->MajorFunction, MajorFunctionToString(stack->MajorFunction));
```

Вспомогательная функция `MajorFunctionToString` возвращает строковое представление кода первичной функции. Например, для запроса `IRP_MJ_READ` она возвращает строку «IRP_MJ_READ».

В этот момент драйвер может продолжить анализ запроса. Если был получен запрос IRP_MJ_DEVICE_CONTROL, он может проверить код управляющей операции и входной буфер. Для запроса IRP_MJ_WRITE он может проверить пользовательский буфер и т. д.

> Драйвер можно расширить, чтобы он сохранял эти запросы в некотором списке (как это делалось в главах 8 и 9, например); затем клиент пользовательского режима будет запрашивать сохраненную информацию. Читатель может реализовать эту возможность самостоятельно.

Наконец, мы не хотим нарушить работоспособность целевого устройства, поэтому запрос передается далее в неизменном виде:

```
    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(ext->LowerDeviceObject, Irp);
}
```

Функция DevMonDeviceControl, упоминавшаяся ранее, является обработчиком драйвера для запросов IRP_MJ_DEVICE_CONTROL. Она используется для динамического добавления или удаления устройств из цепочки фильтрации. Определены следующие коды управляющих операций (из файла KDevMonCommon.h):

```
#define IOCTL_DEVMON_ADD_DEVICE \
    CTL_CODE(0x8000, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_REMOVE_DEVICE \
    CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_REMOVE_ALL \
    CTL_CODE(0x8000, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Вероятно, к этому моменту код обработки будет достаточно понятным:

```
NTSTATUS DevMonDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    auto code = stack->Parameters.DeviceIoControl.IoControlCode;

    switch (code) {
        case IOCTL_DEVMON_ADD_DEVICE:
        case IOCTL_DEVMON_REMOVE_DEVICE:
        {
            auto buffer = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
            auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
            if (buffer == nullptr || len < 2 || len > 512) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }

            buffer[len / sizeof(WCHAR) - 1] = L'\0';
            if (code == IOCTL_DEVMON_ADD_DEVICE)
                status = g_Data.AddDevice(buffer);
```

```
            else {
                auto removed = g_Data.RemoveDevice(buffer);
                status = removed ? STATUS_SUCCESS : STATUS_NOT_FOUND;
            }
            break;
        }
        case IOCTL_DEVMON_REMOVE_ALL:
        {
            g_Data.RemoveAllDevices();
            status = STATUS_SUCCESS;
            break;
        }
    }

    return CompleteRequest(Irp, status);
}
```

Тестирование драйвера

Консольное приложение пользовательского режима снова получается довольно стандартным: оно получает команды для добавления и удаления устройств. Несколько примеров ввода команд:

```
devmon add \device\procexp152
devmon remove \device\procexp152
devmon clear
```

Функция `main` клиента пользовательского режима (с минимальной обработкой ошибок):

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2)
        return Usage();

    auto& cmd = argv[1];

    HANDLE hDevice = ::CreateFile(L"\\\\.\\kdevmon", GENERIC_READ | \
GENERIC_WRITE,
        FILE_SHARE_READ, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("Failed to open device");

    DWORD bytes;
    if (::_wcsicmp(cmd, L"add") == 0) {
        if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_ADD_DEVICE, (PVOID)argv[2],
            static_cast<DWORD>(::wcslen(argv[2]) + 1) * sizeof(WCHAR), nullptr,
0,
            &bytes, nullptr))
            return Error("Failed in add device");
        printf("Add device %ws successful.\n", argv[2]);
        return 0;
    }
```

```
    else if (::_wcsicmp(cmd, L"remove") == 0) {
        if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_DEVICE, (PVOID)
argv[2],
            static_cast<DWORD>(::wcslen(argv[2]) + 1) * sizeof(WCHAR), \
nullptr, 0,
            &bytes, nullptr))
            return Error("Failed in remove device");
        printf("Remove device %ws successful.\n", argv[2]);
        return 0;
    }
    else if (::_wcsicmp(cmd, L"clear") == 0) {
        if (!::DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_ALL,
            nullptr, 0, nullptr, 0, &bytes, nullptr))
            return Error("Failed in remove all devices");
        printf("Removed all devices successful.\n");
    }
    else {
        printf("Unknown command.\n");
        return Usage();
    }

    return 0;
}
```

Подобный код уже неоднократно встречался вам ранее.

Пример команды установки драйвера:

```
sc create devmon type= kernel binpath= c:\book\kdevmon.sys
```

Пример команды запуска:

```
sc start devmon
```

В первом примере запустим Process Explorer (программа должна выполняться с повышенными привилегиями, чтобы ее драйвер мог быть установлен в случае необходимости) и включим фильтрацию поступающих запросов:

```
devmon add \device\procexp152
```

Напомню, что в WinObj выводится устройство с именем ProcExp152 в каталоге Device пространства имен диспетчера объектов. Запустите программу DbgView из пакета SysInternals с повышенными привилегиями и настройте ее для сохранения вывода режима ядра в журнале. Пример вывода:

```
1 0.00000000 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 \
(IRP_MJ_DEVICE_CONTROL)
2 0.00016690 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 \
(IRP_MJ_DEVICE_CONTROL)
```

```
3 0.00041660 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 \
(IRP_MJ_DEVICE_CONTROL)
4 0.00058020 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 \
(IRP_MJ_DEVICE_CONTROL)
5 0.00071720 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14
                                         (IRP_MJ_DEVICE_\
CONTROL)
```

Вероятно, вас не удивит, что Process Explorer на этой машине имеет идентификатор процесса 5432 (и содержит поток с идентификатором 8820). Очевидно, Process Explorer регулярно отправляет своему драйверу запросы, и мы видим, что это всегда запросы `IRP_MJ_DEVICE_CONTROL`.

Устройства, для которых возможна фильтрация, просматриваются в программе WinObj; прежде всего это устройства из каталога Device (рис. 11.14).

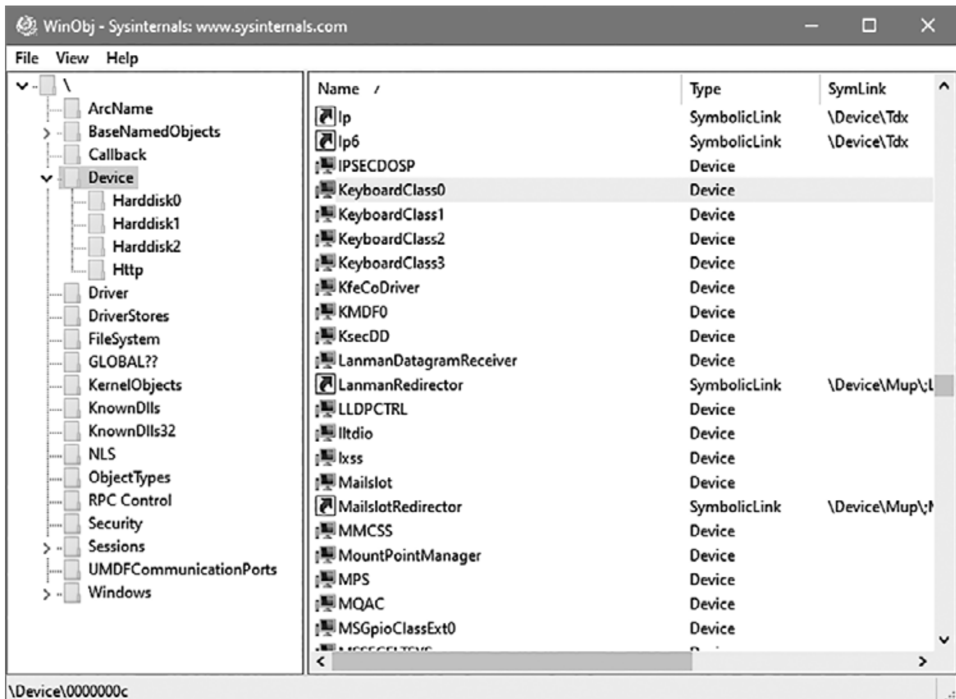


Рис. 11.14. Каталог Device в WinObj

Включим фильтрацию для устройства `keyboardclass0`, находящегося под управлением драйвера класса клавиатуры:

```
devmon add \device\keyboardclass0
```

Теперь при нажатии клавиш для каждой клавиши вы будете получать строку вывода. Результат выглядит примерно так:

```
1 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
2 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
3 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
4 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
5 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
6 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
```

Что такое процесс 612? Это экземпляр CSrss.exe, выполняемый в сеансе пользователя. Одна из задач CSrss — получение данных от устройств ввода. Следует заметить, что это операция чтения, а следовательно, от драйвера класса клавиатуры потребуется некоторый буфер ответа. Но как получить его? Ответ на этот вопрос будет дан в следующем сеансе.

Вы можете опробовать другие устройства. Для одних устройств попытки присоединения могут оказаться неудачными (обычно для устройств, открытых для монопольного доступа), другие могут не подходить для фильтрации такого рода (прежде всего драйвера файловой системы).

Пример для устройства MUP (Multiple UNC Provider):

```
devmon add \device\mup
```

Перейдите в какую-нибудь сетевую папку; в результатах наблюдается довольно серьезная активность:

```
001 11:46:19 driver: \FileSystem\FltMgr: PID: 4, TID: 6236, MJ=2 (IRP_MJ_CLOSE)
002 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=0
                                         (IRP_MJ_CREATE)
003 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=13
                                         (IRP_MJ_FILE_SY\
STEM_CONTROL)
004 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=18
                                         (IRP_MJ_CLEANUP\
)
005 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=2
                                         (IRP_MJ_CLOSE)
006 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=0
                                         (IRP_MJ_CREATE)
007 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=13
                                         (IRP_MJ_FILE_SY\
STEM_CONTROL)
...
054 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=13
                                         (IRP_MJ_FILE_SY\
STEM_CONTROL)
055 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=18
                                         (IRP_MJ_CLEANUP\
)
```

```
056 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=2
                                         (IRP_MJ_CLOSE)
057 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=5
                                         (IRP_MJ_QUERY_IN\
FORMATION)
...
094 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=0
                                         (IRP_MJ_CREATE)
095 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=0
                                         (IRP_MJ_CREATE)
096 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=5
                                         (IRP_MJ_QUERY_IN\
FORMATION)
097 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=18
                                         (IRP_MJ_CLEANUP\
)
098 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=5
                                         (IRP_MJ_QUERY_IN\
FORMATION)
099 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=2
                                         (IRP_MJ_CLOSE)
100 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=12
                                         (IRP_MJ_DIRECTO\
RY_CONTROL)
101 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=0
                                         (IRP_MJ_CREATE)
102 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=12
                                         (IRP_MJ_DIRECTO\
RY_CONTROL)
103 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=18
                                         (IRP_MJ_CLEANUP\
)
104 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=2
                                         (IRP_MJ_CLOSE)
105 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=5
                                         (IRP_MJ_QUERY_IN\
FORMATION)
106 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=12
                                         (IRP_MJ_DIRECTO\
RY_CONTROL)
107 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=27 (IRP_MJ_PNP)
```

Обратите внимание на иерархию над диспетчером фильтров (см. главу 10). Также следует заметить, что в происходящем участвуют несколько процессов (все являются экземплярами Explorer.exe). Устройство MUP — том для удаленной файловой системы. Для фильтрации устройств такого типа лучше использовать мини-фильтры файловой системы.

Не стесняйтесь, экспериментируйте!

Результаты запросов

Обобщенный обработчик в драйвере DevMon видит только входящие запросы. Мы можем их анализировать, но остается интересный вопрос: как получить результаты запроса? Какой-то драйвер в стеке устройства вызовет `IoCompleteRequest`. Если результат представляет интерес для драйвера, он должен определить функцию завершения ввода/вывода.

Как обсуждалось в главе 7, функции завершения при вызове `IoCompleteRequest` вызываются в порядке, обратном порядку регистрации. Каждый уровень в стеке устройства (кроме самого нижнего) может назначить функцию завершения, которая должна вызываться в составе завершения запроса. В этот момент драйвер может проанализировать статус IRP, проверить выходные буферы и т. д.

Функция завершения назначается вызовом `IoSetCompletionRoutine` или (лучше) `IoSetCompletionRoutineEx`. Прототип второй функции выглядит так:

```
NTSTATUS IoSetCompletionRoutineEx (
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_ PIRP Irp,
    _In_ PIO_COMPLETION_ROUTINE CompletionRoutine,
    _In_opt_ PVOID Context, // Определяется драйвером
    _In_ BOOLEAN InvokeOnSuccess,
    _In_ BOOLEAN InvokeOnError,
    _In_ BOOLEAN InvokeOnCancel);
```

Параметры в основном понятны без объяснений. Последние три параметра указывают, для какого статуса завершения IRP должна вызываться функция завершения:

- Если параметр `InvokeOnSuccess` равен `TRUE`, то функция завершения вызывается в том случае, если статус IRP проходит проверку макросом `NT_SUCCESS`.
- Если параметр `InvokeOnError` равен `TRUE`, то функция завершения вызывается в том случае, если статус IRP не проходит проверку макросом `NT_SUCCESS`.
- Если параметр `InvokeOnCancel` равен `TRUE`, то функция завершения вызывается в том случае, если статус IRP равен `STATUS_CANCELLED` (признак того, что запрос был отменен).

Сама функция завершения должна иметь следующий прототип:

```
NTSTATUS CompletionRoutine (
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_ PIRP Irp,
    _In_opt_ PVOID Context);
```

Функция завершения вызывается произвольным потоком (тем, который вызвал `IoCompleteRequest`) на уровне IRQL <= `DISPATCH_LEVEL` (2). А это означает, что должны выполняться все правила для IRQL 2, перечисленные в главе 6.

Что может сделать функция завершения? Она может проанализировать статус IRP и буферы, а также вызвать `IoGetCurrentIrpStackLocation` для получения дополнительной информации из `IO_STACK_LOCATION`. Она не должна вызывать `IoCompleteRequest`, потому что это уже произошло (собственно, именно из-за этого мы уже находимся в функции завершения).

Как насчет возвращаемого статуса? Возможны только два варианта: `STATUS_MORE_PROCESSING_REQUIRED` и все остальное. При возвращении этого специального статуса диспетчер ввода/вывода останавливает распространение IRP вверх по стеку устройства и отменяет факт завершения IRP. Таким образом драйвер становится владельцем IRP и должен со временем снова вызвать `IoCompleteRequest` (что не является ошибкой). Этот вариант чаще всего встречается в драйверах физических устройств, в книге он не рассматривается.

С любыми другими статусами, возвращаемыми из функции завершения, IRP продолжает распространяться вверх по стеку устройства, что может сопровождаться вызовом других функций завершения для драйверов верхних уровней. В этом случае драйвер должен пометить запрос IRP как незавершенный, если устройство более низкого уровня пометило его таковым:

```
if (Irp->PendingReturned)
    IoMarkIrpPending(Irp); // Назначает SL_PENDING_RETURNED
                           // в irpStackLoc->Control
```

Это необходимо из-за того, что диспетчер ввода/вывода делает следующее после возврата управления из функции завершения:

```
Irp->PendingReturned = irpStackLoc->Control & SL_PENDING_RETURNED;
```

Конкретные причины для всех этих нюансов выходят за рамки книги. Лучший источник информации по этой теме — отличная книга Уолтера Уани (Walter Oney) «Programming the Windows Driver Model», второе издание (MS Press, 2003). И хотя книга достаточно старая (в ней рассматривается Windows XP), (а еще в ней рассматриваются только драйверы физических устройств), материал остается актуальным и в ней можно найти много полезного.

Реализуйте функцию завершения ввода/вывода для драйвера DevMon.

Перехват операций драйверов

Использование драйверов фильтров, описанных в этой главе и в главе 10, открывает замечательные возможности для разработчиков драйверов: возможность перехвата запросов практически к любому устройству. В этом разделе я хотел бы упомянуть другой метод — хотя он и не считается «официальным», он может быть полезным в некоторых случаях.

Механизм перехвата операций драйверов основан на идее замены указателей на функции диспетчеризации работающих драйверов. Он автоматически предоставляет возможность «фильтрации» для всех устройств, которыми управляет данный драйвер. Перехватывающий драйвер сохраняет старые указатели на функции, после чего заменяет массив первичных функций в объекте драйвера своими собственными функциями. Теперь любой запрос к устройству, находящемуся под управлением перехватываемого драйвера, будет активизировать функции диспетчеризации перехватывающего драйвера. Никакие дополнительные объекты устройств и присоединение в этой схеме не задействованы.

Механизм `PatchGuard` защищает некоторые драйверы от подобных перехватчиков. Классическим примером служит драйвер файловой системы NTFS — в Windows 8 и выше подобный перехват невозможен. А если бы это было возможно, то фатальный сбой системы был бы делом максимум нескольких минут.

`PatchGuard` (также Kernel Patch Protection) — механизм защиты ядра, который хеширует все структуры данных, которые считаются важными, и при обнаружении любых изменений активизирует фатальный сбой системы. Классический пример — таблица SSDT (System Service Dispatch Table), содержащая указатели на различные системные сервисные функции. Начиная с Vista (64-разрядная версия), перехват запросов к этой таблице невозможен.

У драйверов есть имена, которые являются частью пространства имен диспетчера объектов; они находятся в каталоге `Driver`. На рис. 11.15 показано содержимое каталога в WinObj (программа должна быть запущена с повышенными привилегиями).

Чтобы организовать перехват запросов к драйверу, необходимо найти указатель на объект устройства (`DRIVER_OBJECT`). А для этого необходимо использовать недокументированную, но экспортируемую функцию, которая находит любой объект по имени:

```
NTSTATUS ObReferenceObjectByName (
    _In_ PUNICODE_STRING ObjectPath,
    _In_ ULONG Attributes,
    _In_opt_ PACCESS_STATE PassedAccessState,
    _In_opt_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_TYPE ObjectType,
    _In_ KPROCESSOR_MODE AccessMode,
    _Inout_opt_ PVOID ParseContext,
    _Out_ PVOID *Object);
```

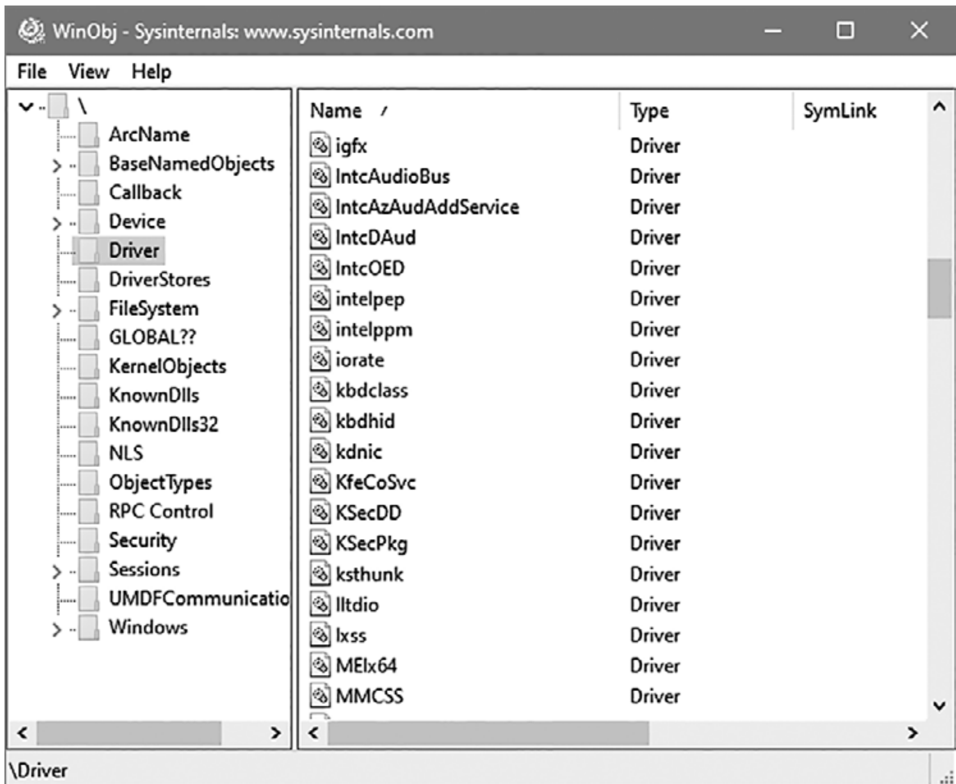


Рис. 11.15. Каталог Driver в WinObj

Пример вызова `ObReferenceObjectByName` для нахождения драйвера `kbdclass`:

```
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\driver\\kbdclass");

PDRIVER_OBJECT driver;
```

```
auto status = ObReferenceObjectByName(&name, OBJ_CASE_INSENSITIVE,
    nullptr, 0, *IoDriverObjectType, KernelMode,
    nullptr, (PVOID*)&driver);
if(NT_SUCCESS(status)) {
    // Работа с драйвером
    ObDereferenceObject(driver); // С течением времени
}
```

Теперь перехватывающий драйвер может заменять указатели на первичные функции, функцию выгрузки, функцию добавления устройства и т. д. При любой замене такого рода предыдущие указатели на функции всегда должны сохраняться для восстановления при отмене перехвата и для перенаправления запроса реальному драйверу. Так как эта замена должна происходить атомарно, лучше использовать `InterlockedExchangePointer`.

Следующий фрагмент кода показывает, как это делается:

```
for (int j = 0; j <= IRP_MJ_MAXIMUM_FUNCTION; j++) {
    InterlockedExchangePointer((PVOID*)&driver->MajorFunction[j], MyHookDispatch);
}
InterlockedExchangePointer((PVOID*)&driver->DriverUnload, MyHookUnload);
```

> Довольно подробный пример метода перехвата можно найти в моем проекте DriverMon на Github по адресу https://github.com/zodiacon/DriverMon.

Библиотеки режима ядра

В процессе написания драйверов мы разработали некоторые классы и вспомогательные функции, которые могут использоваться в разных драйверах. Будет разумно упаковать их в одну библиотеку, которую можно было бы использовать вместо копирования исходных файлов из проекта в проект.

Шаблоны проектов, включенные в WDK, не предоставляют статическую библиотеку для драйверов, но создать такую библиотеку относительно несложно. Для этого можно создать нормальный проект драйвера (например, на основе шаблона WDM Empty Driver), а затем заменить тип проекта на статическую библиотеку, как показано на рис. 11.16.

Если вы захотите включить эту библиотеку в проект драйвера, достаточно добавить ссылку в Visual Studio: щелкните правой кнопкой мыши на разделе References на панели Solution Explorer, выберите команду Add Reference... и выберите проект библиотеки. На рис. 11.17 показан раздел References драйвера после добавления ссылки.

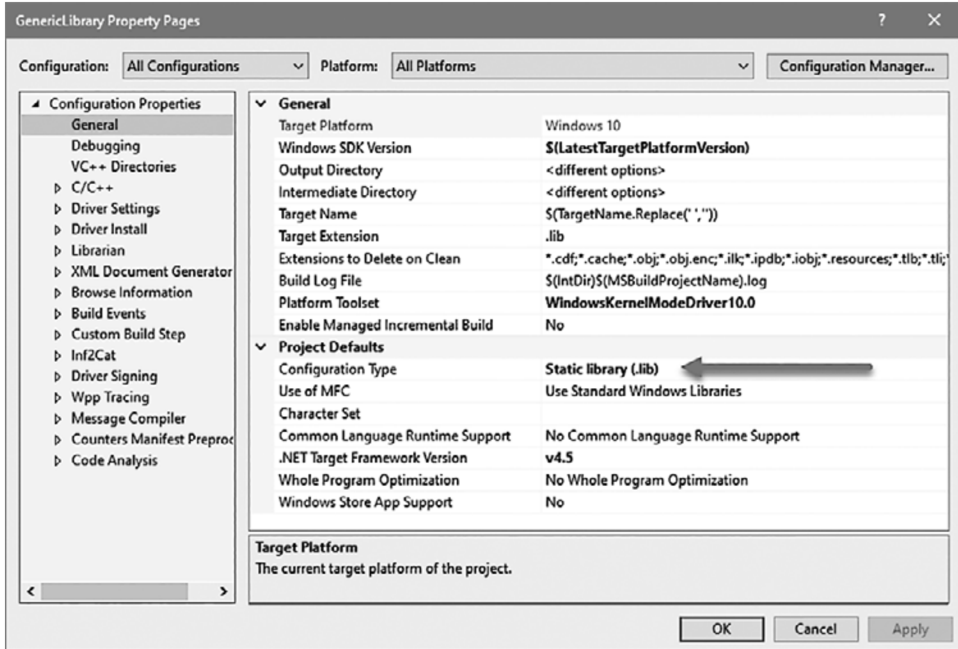


Рис. 11.16. Настройка статической библиотеки режима ядра

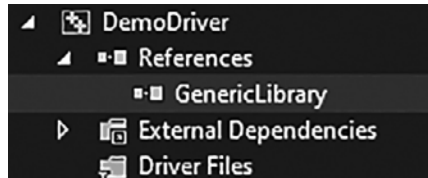


Рис. 11.17. Включение ссылки на библиотеку

> То же самое можно сделать в «старом стиле»: добавьте файл LIB на вход компоновщика с использованием свойств проекта или включите директиву `#pragma comment(lib, "genericlibrary.lib")` в исходный файл.

Итоги

В этой книге обсуждается множество вопросов, так как тема драйверов режима ядра чрезвычайно обширна. Тем не менее книгу следует рассматривать как вводный учебник в мир драйверов устройств режима ядра. Некоторые из тем, которые в книге не рассматривались:

- Драйверы физических устройств.
- Сетевые драйверы и фильтры.
- WFP (Windows Filtering Platform).
- Более подробная информация о мини-фильтрах файловой системы.
- Другие общие средства разработки: таблицы хранения данных, AVL-деревья, битовые карты.
- Типы драйверов для конкретных технологий: HID (Human Interface Device), экран, звуковая карта, Bluetooth, хранение данных...

Некоторые из этих тем могли бы стать хорошими кандидатами для будущей книги более высокого уровня.

Компания Microsoft документировала все описанные типы драйверов; также на сайте Github доступны регулярно обновляемые примеры. Они должны стать отправной точкой для поиска дополнительной информации.

На этом книга подошла к концу. Желаю приятного программирования режима ядра!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Павел Йосифович

Работа с ядром Windows

Перевел с английского Е. Матвеев

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Петруненко*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Викторова, М. Молчанова*
Верстка	*Л. Соловьева*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2021.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.02.21. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 32,250. Тираж 500. Заказ 0000.

Дэвид Калавера, Лоренцо Фонтана

BPF для мониторинга Linux

Виртуальная машина BPF — один из важнейших компонентов ядра Linux. Ее грамотное применение позволит системным инженерам находить сбои и решать даже самые сложные проблемы.

Вы научитесь создавать программы, отслеживающие и модифицирующие поведение ядра, сможете безопасно внедрять код для наблюдения событий в ядре и многое другое.

Дэвид Калавера и Лоренцо Фонтана помогут вам раскрыть возможности BPF. Расширьте свои знания об оптимизации производительности, сетях, безопасности.

Используйте BPF для отслеживания и модификации поведения ядра Linux.

Внедряйте код для безопасного мониторинга событий в ядре — без необходимости перекомпилировать ядро или перезагружать систему.

Пользуйтесь удобными примерами кода на C, Go или Python.

Управляйте ситуацией, владея жизненным циклом программы BPF.

Пол Тронкон, Карл Олбинг

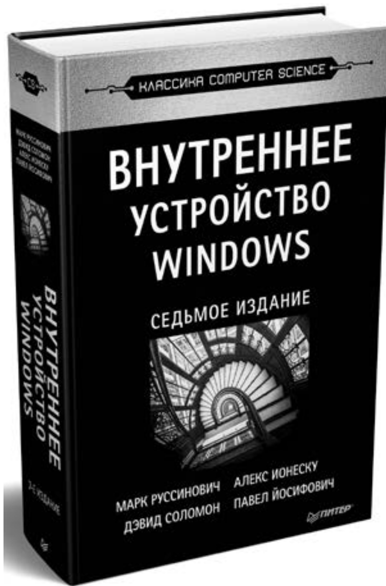
Bash и кибербезопасность:
атака, защита и анализ из командной строки Linux

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт.

Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.

Марк Руссинович, Дэвид Соломон,
Алекс Ионеску, Павел Йосифович

Внутреннее устройство Windows. 7-е изд.



С момента выхода предыдущего издания этой книги операционная система Windows прошла длинный путь обновлений и концептуальных изменений, результатом которых стала новая стабильная архитектура ядра Windows 10.

Книга «Внутреннее устройство Windows» создана для профессионалов, желающих разобраться во внутренней жизни основных компонентов Windows 10. Опираясь на эту информацию, разработчикам будет проще находить правильные проектные решения, создавая приложения для платформы Windows, и решать сложные проблемы, связанные с их эксплуатацией. Системные администраторы, зная что находится у операционной системы «под капотом», смогут разобраться с поведением системы и быстрее решать задачи повышения производительности и диагностики сбоев. Специалистам по безопасности пригодится информация о борьбе с уязвимостями операционной системы.

Прочитав эту книгу, вы будете лучше разбираться в работе Windows и в истинных причинах того или иного поведения ОС.