

## Глава 3. Основы программирования ядра windows

Предыдущие версии глав:

<https://ru-sfera.org/threads/windows-kernel-programming-glava-1-osnovnye-momenty-i-arxitektura-jadra.3943/>

<https://ru-sfera.org/threads/windows-kernel-programming-glava-2-nachalo-raboty-s-instrumentami-razrabotchika-jadra.3945/>

В этой главе мы углубимся в API, структуры и определения ядра.

Мы также рассмотрим некоторые механизмы, которые вызывают код в драйвере.

Наконец, мы объединим все эти знания, чтобы создать наш первый функциональный драйвер.

**В этой главе:**

Общие рекомендации по программированию ядра.

Отладка и сборки релизов.

API ядра.

Функции и коды ошибок.

Строки.

Динамическое распределение памяти.

Списки.

Объект драйвера.

Объекты устройства.

**Итак начнем:**

### **1) Общие рекомендации по программированию ядра**

Для разработки драйверов ядра требуется Windows Driver Kit (WDK), где находятся соответствующие заголовки и необходимые библиотеки.

API ядра состоит из функций, написанных на языке «С», очень похожих по сути на разработку пользовательского режима.

Однако есть несколько отличий. Таблица 3-1 показывает важные различия между программированием в пользовательском режиме и программированием в режиме ядра.

|                           | <b>User Mode</b>   | <b>Kernel Mode</b>   |
|---------------------------|--|--|
| Необработанное исключение | Вызывает крах процесса.  | Вызывает крах всей системы.  |
| Завершение работы         | Когда процесс завершается, приватная память и ресурсы освобождаются автоматически. | Если драйвер выгружается без освобождения, то все - что он использовал, не будет выгружено автоматически и может привести к утечки ресурсов. |
| Возвращаемые значения IRQ | Ошибки API иногда игнорируются.  | Почти никогда не игнорируются ошибки.  |
| Плохой код                | Эффект в пределах процесса.  | Эффект в пределах всей системы.  |
| Тестирование и отладка    | Типичное тестирование и отладка выполняются на машина разработчика.                | Часто отладка выполняется на сторонней машине.   |
| Библиотеки                | Можно использовать C/C++ библиотеки (Например Boost, STL).                         | Большинство стандартных библиотек не могут быть использованы   |
| Обработка исключений      | Можно использовать исключения C++ или структурированную обработку исключений (SEH) | Можно использовать только SEH  |
| Использование C++         | Полная версия C++ runtime.   | Нет C++ runtime.   |

## **Необработанные исключения**

Исключения, возникающие в пользовательском режиме, которые не перехвачены программой, приводят к завершению процесса преждевременно (Например переполнение буфера и т.д.).

Код режима ядра, с другой стороны, будучи доверенным, не может восстановиться после необработанного исключения.

Такое исключение вызывает сбой системы с печально известным синим экраном смерти (BSOD) (более новые версии Windows имеют более разнообразные цвета для аварийного экрана).

Поначалу BSOD может показаться наказанием, но по сути это защитный механизм.

Рациональный в то время как продолжение выполнения кода может привести к необратимому повреждению Windows (например, удаление важных файлов или повреждение реестра), которые могут привести к сбою загрузки системы.

Поэтому лучше немедленно все остановить, чтобы предотвратить потенциальный ущерб. Мы обсудим BSOD более подробно в главе 6.

Все это приводит по крайней мере к одному простому выводу: код ядра должен быть тщательно закодирован, тщательно и не пропуская какие-либо детали или проверки ошибок.

## **Завершение работы**

Когда процесс завершается по какой-либо причине, то вся личная память освобождается, все дескрипторы закрываются и т. д.

Конечно, преждевременное закрытие дескриптора может привести к некоторой потере данных, например дескриптор файла закрывается перед сбросом некоторых данных на диск - но утечек нет, это гарантируется ядром.

Драйверы ядра, с другой стороны, не дают такой гарантии.

Если драйвер выгружается, в это время все еще будет выделена память или открыты дескрипторы ядра - эти ресурсы не будут освобождены автоматически, а будут освобождены только при следующей загрузке системы.

**Почему так ?** Ядро не может отслеживать распределение драйверов и использование ресурсов, поэтому они не могут быть освобождены автоматически при выгрузке драйвера.

Теоретически этого было бы возможно достичь (хотя в настоящее время ядро не отслеживает такое использование ресурса).

Реальная проблема заключается в том, что ядру было бы слишком опасно пытаться очистить ресурсы драйвера.

В ядре нет способа узнать, утекли-ли ресурсы драйвера по какой-то причине.

Например, драйвер может выделить некоторый буфер и затем передать его другому драйверу.

Этот второй драйвер может использовать буфер памяти и в конце концов освободить его.

Если ядро попытается освободить буфер при выгрузке первого драйвера, второй драйвер вызовет нарушение при доступе к этому теперь освобожденному буферу, и в итоге вызовет сбой системы.

Это еще раз подчеркивает ответственность драйвера ядра за правильную очистку после себя; никто остальной не сделает это.

### **Возвращаемое значение функции**

В типичном коде пользовательского режима возвращаемые значения из функций API иногда игнорируются, разработчик будучи несколько оптимистичным, что вызванная функция вряд ли потерпит неудачу.

В худшем случае необработанное исключение позже завершит процесс; система, однако, остается нетронутой.

Игнорирование возвращаемых значений из API ядра намного опаснее, и вообще следует избегать. Даже на первый взгляд «невинные» функции могут потерпеть неудачу, поэтому золотое правило здесь - **всегда проверяйте возвращаемые значения статуса из API ядра.**

### **IRQL**

Interrupt Request Level (IRQL) является важной концепцией ядра, которая будет более подробно рассмотрена в главе 6.

Достаточно сказать, что обычно IRQL процессора равен нулю, а точнее, это всегда ноль, когда код пользовательского режима выполняется.

В режиме ядра большую часть времени он все еще равен нулю, но не все время.

Эффекты IRQL выше нуля будут обсуждаться в главе 6.

## Использование C++

В программировании пользовательского режима C++ использовался много лет, и он хорошо работает в сочетании с вызовами API пользовательского режима.

C кодом ядра Microsoft начала официально поддерживать C++ с Visual Studio 2012 и WDK 8.

C++, конечно, не является обязательным, но имеет ряд важных преимуществ, связанных с очисткой ресурсов, используя идиому C++ под названием Resource Acquisition Is Initialization (RAII).

C++ как язык почти полностью поддерживается для кода ядра.

Но в C++ нет времени исполнения в ядре, и поэтому некоторые функции C++ просто не могут быть использованы:

- Операторы new и delete не поддерживаются и не будут компилироваться. Это потому что их нормальная работа - выделять из кучи пользовательского режима, что, конечно, невозможно в ядре.

API ядра имеет функции «замены», malloc и free. Мы обсудим эти функции позже в этой главе.

Возможно, однако перегрузить эти операторы аналогично тому, как это делается в пользовательском режиме C++, и вызвать распределение ядра и свободные функции. Мы увидим, как это сделать позже в этой главе.

- Глобальные переменные, которые имеют конструкторы, отличные от заданных по умолчанию, не будут вызываться. Этим ситуациям можно избежать несколькими способами:

- Избегайте любого кода в конструкторе и вместо этого создайте некоторую функцию Init для вызова явно из кода драйвера (например, из DriverEntry).

- Выделите указатель только как глобальную переменную и динамически создайте фактический экземпляр.

Компилятор сгенерирует правильный код для вызова конструктора. Это работает при условии, если операторы new и delete были перегружены, как описано далее в этой главе.

- Ключевые слова обработки исключений в C++ (try, catch, throw) не компилируются. Это потому что механизм обработки исключений C++ требует собственной среды выполнения, которой нет в ядре.

Обработка исключений может быть выполнена только с использованием структурированной обработки исключений (SEH). Мы подробно рассмотрим SEH в главе 6.

- Стандартные библиотеки C++ недоступны в ядре. Хотя большинство из них основано на шаблонах, они не компилируются, потому что они зависят от библиотек пользовательского режима и семантики.

Шаблоны C++ как языковая функция прекрасно работают и могут быть использованы, например, для создания альтернативных типов для типов библиотеки пользовательского режима, такие как `std::vector`, `std::wstring` и т. д.

Примеры кода в этой книге используют C++.

Функции, в основном используемые в примерах используют следующие типы:

- Ключевое слово `nullptr`, представляющее истинный указатель `NULL`.
- Ключевое слово `auto`, позволяющее вывести тип при объявлении и инициализации переменных.

Это полезно для уменьшения беспорядка, и сосредоточения внимания на важных деталях.

- Шаблоны будут использоваться, когда они имеют смысл.
- Перегрузка новых операторов.
- Конструкторы и деструкторы, особенно для построения типов RAII.

Строго говоря, драйверы могут быть написаны на чистом C без каких-либо проблем. Если вы предпочитаете идти по этому пути, используйте файлы с расширением C, а не CPP. Это автоматически вызовет компилятор Си.

## Тестирование и отладка

При использовании кода пользовательского режима тестирование обычно выполняется на компьютере разработчика (если все требуемые зависимости могут быть удовлетворены).

Отладка обычно выполняется путем подключения отладчика (Visual Studio в большинстве случаев) к запущенному процессу (или процессам).

При использовании кода ядра тестирование обычно выполняется на другой машине, обычно на виртуальной машине, размещенной на машине разработчика. Это гарантирует, что в случае BSOD машина разработчика не пострадает.

Отладка кода ядра должна выполняться на другом компьютере, на котором выполняется настоящий драйвер.

Это связано с тем, что в режиме ядра попадание на точку останова останавливает всю машину, а не только конкретный процесс.

Это означает, что на машине разработчика находится сам отладчик, а на второй машине (опять же, как правило, виртуальная машина) выполняет код драйвера.

Эти две машины должны быть связаны через некоторый механизм, чтобы данные могли проходить между хостом (где работает отладчик) и таргетом. Мы рассмотрим отладку ядра более подробно в главе 5.

## **Отладка и сборка проекта**

Как и в проектах пользовательского режима, сборка драйверов ядра может выполняться в режиме отладки или релиза.

Различия схожи с аналогами в пользовательском режиме - отладочная сборка не требует оптимизации по умолчанию, но проще в отладке.

В сборках релизов используется оптимизация компилятора для создания самого быстрого кода. Однако есть несколько отличий.

Действительными терминами в терминологии ядра являются Checked (Debug) и Free (Release).

Хотя Visual Studio продолжает использовать термины Debug/Release, более старая документация использует Debug/free.

С точки зрения компиляции, сборки отладки ядра определяют дефайн DBG и устанавливают его значение равным 1 (по сравнению с дефайном \_DEBUG, определенным в режиме пользователя). Это означает, что вы можете использовать дефайн DBG, чтобы различать сборки Debug и Release с условной компиляцией.

Фактически это то, что делает макрос KdPrint: в сборках Debug он компилируется в вызов DbgPrint, а в сборке Release он компилируется в пустую строку, в результате чего вызовы KdPrint не влияют на сборку Release.

## API ядра

Драйверы ядра используют экспортированные функции из компонентов ядра. Эти функции будут называться API ядра. Большинство функций реализовано в самом модуле ядра (NtOskrnl.exe), но некоторые могут быть реализованы другими модулями ядра, такими как HAL (hal.dll).

Kernel API - это большой набор функций языка Си. Большинство из этих функций начинаются с префикса, компонента, реализующий эту функцию.

В таблице 3-2 приведены некоторые распространенные префиксы и их смысл:

| Префикс | Описание   | Пример                     |
|---------|--|----------------------------|
| Ex      | Общие исполнительные функции                     | ExAllocatePool             |
| Ke      | Общие функции ядра                               | KeAcquireSpinLock          |
| Mm      | Менеджер памяти                                  | MmProbeAndLockPages        |
| Rtl     | Функции runtime library (Обработка строк и т.д.) | RtlInitUnicodeString       |
| FsRtl   | Функции файловой системы                         | FsRtlGetFileSize           |
| Flt     | Функции мини-филтра                              | FltCreateFile              |
| Ob      | Менеджер объектов                                | ObReferenceObject          |
| Io      | Менеджер I/O                                     | IoCompleteRequest          |
| Se      | security   | SeAccessCheck              |
| Ps      | Функции работы со структурами процесса           | PsLookupProcessByProcessId |
| Po      | Менеджер питания                                 | PoSetSystemState           |
| Wmi     | Инструментарий управления Windows                | WmiTraceMessage            |
| Zw      | Обертки API                                      | ZwCreateFile               |
| Cm      | Менеджер конфигурации                            | CmRegisterCallbackEx       |
| Hal     | Уровень аппаратных абстракций                    | HalExamineMBR              |



Если вы посмотрите на список экспортируемых функций из NtOsKrn1.exe, вы найдете больше функций, которые фактически задокументированы в комплекте драйверов Windows; это просто факт жизни разработчика ядра - не все задокументировано.

На этом этапе обсуждается один набор функций - функции с префиксом Zw.

Эти функции зеркальное отображение собственных API-интерфейсов, доступных в виде шлюзов из NtDll.Dll (Обертки).

Когда функция Nt вызывается из пользовательского режима, как например NtCreateFile, она фактически вызывает NtCreateFile в ядерном режиме.

При этом NtCreateFile может сделать различные проверки, основанные на том, что исходный вызывающий абонент находится в режиме пользователя.

Эта информация вызывающей стороны хранится по отдельности, в недокументированном элементе PreviousMode в KTHREAD структуры для каждого потока.

С другой стороны, если драйвер ядра должен вызывать системную службу, он не должен подвергаться таким-же проверкам и ограничениям, наложенным на вызывающих абонентов пользовательского режима.

### **Как работают Zw функции:**

Вызов функции Zw устанавливает текущий режим вызова в KernelMode (0), а затем вызывается родная функция (Системный вызов).

Например, вызов ZwCreateFile устанавливает текущего вызывающего в kernelMode и затем вызывает NtCreateFile, заставляя NtCreateFile обойти некоторые проверки безопасности, которые в противном случае необходимо выполнить.

Суть в том, что драйверы ядра должны вызывать функции Zw если нет веской причины поступить иначе.

### **Функции и коды ошибок**

Большинство функций API ядра возвращают статус, указывающий на успех или неудачу операции.

Возвращаемое значение как NTSTATUS, 32-разрядное целое число со знаком.

Значение STATUS\_SUCCESS (0) указывает на успех. Отрицательный значение указывает на какую-то ошибку.

Часто проверку на ошибки можно сделать с помощью макроса NT\_SUCCESS.

Вот пример, который проверяет на неудачу и регистрирует ошибку, если это так:

```
NTSTATUS DoWork() {  
    NTSTATUS status = CallSomeKernelFunction();  
    if(!NT_SUCCESS(status)) {  
        KdPrint((L"Error occurred: 0x%08X\n", status));  
        return status;  
    }  
    // continue with more operations  
    return STATUS_SUCCESS;  
}
```

В некоторых случаях значения NTSTATUS возвращаются из функций, которые в конечном итоге переходят в пользовательский режим.

В этих случаях значение STATUS\_xxx преобразуется в некоторое значение ERROR\_yyy, которое доступно для пользовательский режима через функцию GetLastError.

Обратите внимание, что это не одинаковые цифры. В этом случае это обычно не относится к драйверу ядра.

Внутренние функции драйвера ядра также обычно возвращают NTSTATUS, чтобы указать их успех/неудачу.

Это также подразумевает, что «реальные» возвращаемые значения из функций драйвера обычно возвращаются через указатели или ссылки, представленные в качестве аргументов функции.

## Строки

API ядра использует строки во многих случаях, где это необходимо.

В некоторых случаях эти строки являются простыми Unicode-указателями (`wchar_t *` или один из их typedef-ов, таких как `WCHAR`), но большинство функций работают со строками ожидая структуру типа `UNICODE_STRING`.

**Термин Unicode, используемый в этой книге, примерно эквивалентен UTF-16, что означает 2 байта на символ. Вот так строки хранятся внутри компонентов ядра.**

Структура `UNICODE_STRING` представляет строку с известной длиной и максимальной длиной.

Вот упрощенное определение структуры:

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

Управление структурами `UNICODE_STRING` обычно выполняется с помощью набора функций `Rtl`, которые имеют дело со строками.

Некоторые из общих функций для работы со строками обеспечиваются функциями `Rtl`:

**`RtlInitUnicodeString`** - Инициализирует `UNICODE_STRING` на основе существующей C-строки. Он устанавливает буфер, затем вычисляет длину и устанавливает `MaximumLength` к тому же значению. Обратите внимание, что эта функция не выделяет никакой памяти - она просто инициализирует внутренние элементы.

**`RtlCopyUnicodeString`** - Копирует один `UNICODE_STRING` в другой. Строка назначения (указатель на буфер) должен быть выделен перед копированием и максимальная длина установлена соответствующим образом.

**RtlCompareUnicodeString** - Сравнивает два UNICODE\_STRING (равно, меньше, больше), указывая делать ли сравнение с учетом регистра или без учета регистра.

**RtlEqualUnicodeString** — Сравнивает две строки, на равенство.

**RtlAppendUnicodeStringToString** - Добавляет один UNICODE\_STRING к другому.

**RtlAppendUnicodeToString** - Добавляет UNICODE\_STRING к строке в стиле C.

В дополнение к вышеупомянутым функциям существуют функции, которые работают с указателями на C-строку.

Более того, некоторые из хорошо известных строковых функций из библиотеки C Runtime реализованы в ядре для удобства: wcsncpy, wscat, wcslen, wcsncpy\_s, wcschr, strcpy, strcpy\_s и другие.

Префикс wcs работает со строками C Unicode, а префикс str работает со строками C Ansi.

Суффикс \_s в некоторых функциях указывает на безопасную функцию, где дополнительный аргумент с указанием максимальной длины строки, чтобы функции не передать больше данных, чем этот размер.

### **Динамическое распределение памяти**

Драйверам часто нужно динамически выделять память.

Как обсуждалось в главе 1, размер стека ядра довольно маленький, поэтому любой большой кусок памяти должен выделяться динамически.

Ядро предоставляет два общих пула памяти для использования драйверами (само ядро также использует их).

- Выгружаемый пул - пул памяти, который может быть выгружен при необходимости.
- Non Paged Pool - пул памяти, который никогда не выгружается и гарантированно останется в оперативной памяти.

Ясно, что невыгружаемый пул является «лучшим» пулом памяти, поскольку он никогда не может вызвать сбой страницы.

В некоторых случаях требуется выделение из невыгружаемого пула.

Драйверы должны использовать этот пул экономно, только при необходимости. Во всех остальных случаях драйверы должны использовать выгружаемый пул.

Ниже приведены наиболее полезные функции, используемые для работы с пулами памяти ядра:

**ExAllocatePool** - Выделение памяти из одного из пулов с помощью тега по умолчанию. Эта функция считается устаревшей. Следующая функция должна использоваться вместо этой.

**ExAllocatePoolWithTag** — Выделение памяти из одного из пулов с указанным тегом.

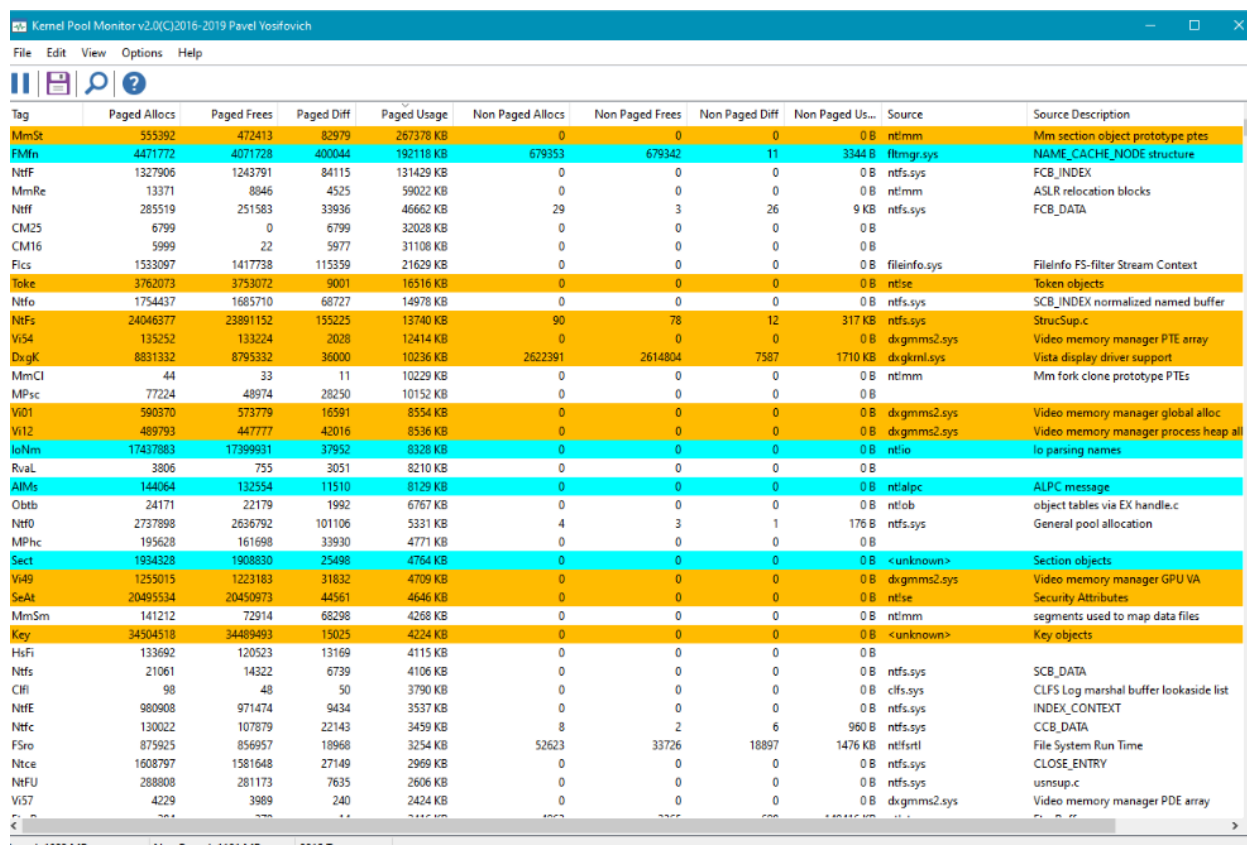
**ExAllocatePoolWithQuotaTag** - Выделение памяти из одного из пулов с указанным тегом и устанавливает квоту текущего процесса для распределения.

**ExFreePool** — Освобождение пула. Функция знает из какого пула сделано выделение.

Аргумент tag в некоторых функциях позволяет именовать выделение памяти 4-байтовым значением логически идентифицирующих драйвер, или некоторую часть.

Эти распределения пула (с их тегами) можно просмотреть с помощью инструмента Poolmon WDK, или мой собственный инструмент PoolMonX (можно загрузить с <http://www.github.com/zodicon/AllTools>).

Рисунок 3-1 показывает снимок экрана PoolMonX (v2).



| Tag  | Paged Allocs | Paged Frees | Paged Diff | Paged Usage | Non Paged Allocs | Non Paged Frees | Non Paged Diff | Non Paged Us... | Source       | Source Description                     |
|------|--------------|-------------|------------|-------------|------------------|-----------------|----------------|-----------------|--------------|--|
| MmSt | 555392       | 472413      | 82979      | 267378 KB   | 0                | 0               | 0              | 0 B             | ntmm         | Mm section object prototype ptes       |
| Fmfn | 4471772      | 4071728     | 400044     | 192118 KB   | 679353           | 679342          | 11             | 3344 B          | ftmgr.sys    | NAME_CACHE_NODE structure              |
| NtF  | 1327906      | 1243791     | 84115      | 131429 KB   | 0                | 0               | 0              | 0 B             | ntfs.sys     | FCB_INDEX                              |
| MmRe | 13371        | 8846        | 4525       | 59022 KB    | 0                | 0               | 0              | 0 B             | ntmm         | ASLR relocation blocks                 |
| Ntff | 285519       | 251583      | 33936      | 46662 KB    | 29               | 3               | 26             | 9 KB            | ntfs.sys     | FCB_DATA                               |
| CM25 | 6799         | 0           | 6799       | 32028 KB    | 0                | 0               | 0              | 0 B             |              |  |
| CM16 | 5999         | 22          | 5977       | 31108 KB    | 0                | 0               | 0              | 0 B             |              |  |
| Fics | 1533097      | 1417738     | 115359     | 21629 KB    | 0                | 0               | 0              | 0 B             | fileinfo.sys | Fileinfo FS-filter Stream Context      |
| Tcke | 3762073      | 3753072     | 9001       | 16516 KB    | 0                | 0               | 0              | 0 B             | ntse         | Token objects                          |
| Ntfo | 1754437      | 1685710     | 68727      | 14978 KB    | 0                | 0               | 0              | 0 B             | ntfs.sys     | SCB_INDEX normalized named buffer      |
| NtFs | 24046377     | 23891152    | 155225     | 13740 KB    | 90               | 78              | 12             | 317 KB          | ntfs.sys     | StrucSup.c                             |
| ViS4 | 135252       | 133224      | 2028       | 12414 KB    | 0                | 0               | 0              | 0 B             | dxgmmms2.sys | Video memory manager PTE array         |
| DvgK | 8831332      | 8795332     | 36000      | 10236 KB    | 2622391          | 2614804         | 7587           | 1710 KB         | dxgkml.sys   | Vista display driver support           |
| MmCl | 44           | 33          | 11         | 10229 KB    | 0                | 0               | 0              | 0 B             | ntmm         | Mm fork clone prototype PTEs           |
| MPsc | 77224        | 48974       | 28250      | 10152 KB    | 0                | 0               | 0              | 0 B             |              |  |
| Vi01 | 590370       | 573779      | 16591      | 8554 KB     | 0                | 0               | 0              | 0 B             | dxgmmms2.sys | Video memory manager global alloc      |
| Vi12 | 489793       | 447777      | 42016      | 8536 KB     | 0                | 0               | 0              | 0 B             | dxgmmms2.sys | Video memory manager process heap all  |
| IoNm | 17437883     | 17399931    | 37952      | 8328 KB     | 0                | 0               | 0              | 0 B             | ntio         | Io parsing names                       |
| Rval | 3806         | 755         | 3051       | 8210 KB     | 0                | 0               | 0              | 0 B             |              |  |
| Alpm | 144064       | 132554      | 11510      | 8129 KB     | 0                | 0               | 0              | 0 B             | ntalpc       | ALPC message                           |
| Obtb | 24171        | 22179       | 1992       | 6767 KB     | 0                | 0               | 0              | 0 B             | ntlob        | object tables via EX handle.c          |
| Ntfo | 2737898      | 2636792     | 101106     | 5331 KB     | 4                | 3               | 1              | 176 B           | ntfs.sys     | General pool allocation                |
| MPsc | 195628       | 161698      | 33930      | 4771 KB     | 0                | 0               | 0              | 0 B             |              |  |
| Sect | 1934328      | 1908830     | 25498      | 4764 KB     | 0                | 0               | 0              | 0 B             | <unknown>    | Section objects                        |
| Vi09 | 1255015      | 1223183     | 31832      | 4709 KB     | 0                | 0               | 0              | 0 B             | dxgmmms2.sys | Video memory manager GPU VA            |
| SeAt | 20495534     | 20459973    | 44561      | 4646 KB     | 0                | 0               | 0              | 0 B             | ntse         | Security Attributes                    |
| MmSm | 141212       | 72914       | 68298      | 4268 KB     | 0                | 0               | 0              | 0 B             | ntmm         | segments used to map data files        |
| Key  | 34504518     | 34489493    | 15025      | 4224 KB     | 0                | 0               | 0              | 0 B             | <unknown>    | Key objects                            |
| HoFi | 133692       | 120523      | 13169      | 4115 KB     | 0                | 0               | 0              | 0 B             |              |  |
| Ntfs | 21061        | 14322       | 6739       | 4106 KB     | 0                | 0               | 0              | 0 B             | ntfs.sys     | SCB_DATA                               |
| Clfi | 98           | 48          | 50         | 3790 KB     | 0                | 0               | 0              | 0 B             | clfs.sys     | CLFS Log marshal buffer lookaside list |
| Ntfe | 980908       | 971474      | 9434       | 3537 KB     | 0                | 0               | 0              | 0 B             | ntfs.sys     | INDEX_CONTEXT                          |
| Ntfc | 130022       | 107879      | 22143      | 3459 KB     | 8                | 2               | 6              | 960 B           | ntfs.sys     | CCB_DATA                               |
| FSro | 875925       | 856957      | 18968      | 3254 KB     | 52623            | 33726           | 18897          | 1476 KB         | ntfsrtl      | File System Run Time                   |
| Ntce | 1608797      | 1581648     | 27149      | 2969 KB     | 0                | 0               | 0              | 0 B             | ntfs.sys     | CLOSE_ENTRY                            |
| NtFU | 288808       | 281173      | 7635       | 2606 KB     | 0                | 0               | 0              | 0 B             | ntfs.sys     | usnup.c                                |
| ViS7 | 4229         | 3989        | 240        | 2424 KB     | 0                | 0               | 0              | 0 B             | dxgmmms2.sys | Video memory manager PDE array         |

В следующем примере кода показано распределение памяти и копирование строки для сохранения пути к реестру.

Строка передается в DriverEntry и освобождение этой строки происходит в подпрограмме Unload:

```
// define a tag (because of little endianess, viewed in PoolMon as '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 is for UNICODE_STRING objects
    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"));
}
```

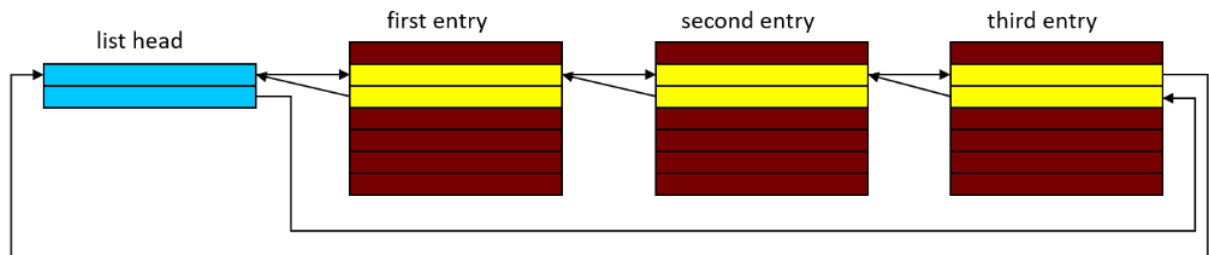
## Списки

Ядро использует круговые двусвязные списки во многих своих внутренних структурах данных.

Например, все процессы в системе управляются структурами EPROCESS, связанными в круговой двойной список, где хранится его голова, в переменной PsActiveProcessHead.

Все эти списки построены аналогичным образом, вокруг структуры LIST\_ENTRY, определенной следующим образом:

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



Одна такая структура встроена в основную структуру.

Например, в EPROCESS структуре, член ActiveProcessLinks имеет тип LIST\_ENTRY, указывая на следующий и предыдущие объект LIST\_ENTRY других структур EPROCESS. Заголовок списка хранится отдельно.

В случае процесса это PsActiveProcessHead.

Получить указатель на фактическую структуру с учетом адреса LIST\_ENTRY можно с помощью макроса CONTAINING\_RECORD.

Например, предположим, что вы хотите управлять списком структур типа MyDataItem, определенных следующим образом:

```
struct MyDataItem {  
    // some data members  
    LIST_ENTRY Link;  
    // more data members  
};
```

При работе с этими связанными списками у нас есть заголовок для списка, который хранится в переменной.

Это означает, что обход осуществляется с помощью члена списка Flink для указания следующего LIST\_ENTRY в списке.

Учитывая указатель на LIST\_ENTRY, мы ищем MyDataItem, который содержит этот элемент списка.

Вот где приходит CONTAINING\_RECORD (Поиск списка):

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

Макрос выполняет правильный расчет смещения и выполняет приведение к фактическому типу данных (MyDataItem в примере).

Ниже приведены функции работы со списками:

***InitializeListHead*** - Инициализирует заголовок списка, чтобы создать пустой список.

***InsertHeadList*** — Вставить элемент в начало списка.

***InsertTailList*** — Вставить элемент в конец списка.

***IsListEmpty*** — Проверка списка на пустоту.

***RemoveHeadList*** - Удалить элемент в начале списка.

***RemoveTailList*** — Удалить элемент в конце списка.

***RemoveEntryList*** - Удалить конкретный элемент из списка.

***ExInterlockedInsertHeadList*** - Вставить элемент в начало списка атомарно, используя указанный спинлок.

***ExInterlockedInsertTailList*** - Вставить элемент в конец списка атомарно, используя указанный спинлок.

***ExInterlockedRemoveHeadList*** - Удалить элемент из начала списка атомарно, используя указанный спинлок.

## **Объекты драйвера**

Мы уже видели, что функция DriverEntry принимает два аргумента, первый - это объект драйвера.

Это частично документированная структура с именем DRIVER\_OBJECT, определенная в WDK заголовке.



«Полу документированный» означает, что некоторые его члены документированы, а некоторые нет.

Эта структура выделяется ядром и частично инициализируется.

Тогда это предоставляется в DriverEntry.

На этом этапе драйвер должен дополнительно инициализировать структуру, чтобы указать, какие операции поддерживаются драйвером.

Одну из таких «операций» мы видели в главе 2 - процедура выгрузки.

Другой важный набор операций для инициализации называется диспетчерскими процедурами.

Это массив указателей функций, член MajorFunction в DRIVER\_OBJECT. Этот набор указывает, какие конкретные операции драйвер поддерживает, например, создание, чтение, запись и т. д. Эти показатели определяются с префиксом IRP\_MJ\_.

Основные коды основных функций и их значение:

**IRP\_MJ\_CREATE (0)** - Создать файл. Обычно вызывается для CreateFile или ZwCreateFile.

**IRP\_MJ\_CLOSE (2)** - Закроить операцию. Обычно вызывается для CloseHandle или ZwClose.

**IRP\_MJ\_READ (3)** - Операция чтения. Обычно вызывается для ReadFile, ZwReadFile и подобные API чтения.

**IRP\_MJ\_WRITE (4)** - Операция записи. Обычно вызывается для WriteFile, ZwWriteFile и подобные API записи.

**IRP\_MJ\_DEVICE\_CONTROL (14)** - Общий вызов драйверу, вызванный вызовами DeviceIoControl или ZwDeviceIoControlFile.

**IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL (15)** - Аналогичен предыдущему, но доступен только для ядра.

**IRP\_MJ\_PNP (31)** - Plug and play обратный вызов, вызванный менеджером Plug and play.

**IRP\_MJ\_POWER (22)** - Power callback, вызываемый Power Manager.

Первоначально массив MajorFunction инициализируется ядром для указания на внутреннюю процедуру ядра IoPInvalidDeviceRequest, который возвращает состояние ошибки вызывающей стороне, указывая, что операция не поддерживается.

Это означает, что драйвер в своей процедуре DriverEntry должен только инициализировать фактические операции, которые он поддерживает, оставляя все остальные записи в их значениях по умолчанию.

Например, наш драйвер Sample на данный момент не поддерживает какие-либо процедуры отправки, что означает нет возможности общаться с драйвером.

Драйвер должен по крайней мере поддерживать IRP\_MJ\_CREATE и операции IRP\_MJ\_CLOSE, чтобы позволить открыть дескриптор для одного объекта устройства драйвера.

Мы воплотим эти идеи в жизнь в следующей главе.

## **Объекты устройства**

Хотя объект драйвера может выглядеть хорошим кандидатом для общения с клиентами, это не так.

Фактические конечные точки связи для клиентов, чтобы общаться с драйверами, являются объектами устройства.

Объекты устройства являются экземплярами полу-документированной структуры DEVICE\_OBJECT.

Драйвер должен создать хотя бы один объект устройства и получить имя, чтобы клиенты могли с ним связаться.

Функция CreateFile (и ее варианты) принимает первый аргумент, который называется «имя файла», но на самом деле это должно указывать на имя объекта устройства, где фактический файл является лишь частным случаем.

Название CreateFile несколько вводит в заблуждение - слово «файл» здесь фактически означает объект файла.

Открытие дескриптора файла или устройства создает экземпляр структуры ядра FILE\_OBJECT.

Точнее, CreateFile принимает символическую ссылку, объект ядра, который знает, как указывать на другой объект ядра. (Вы можете считать символическую ссылку похожей по своей концепции на ярлык в файловой системе.)

Все символические ссылки, которые можно использовать из пользовательского режима CreateFile или CreateFile2 находятся в каталоге диспетчера объектов с именем `??`. Это можно посмотреть с помощью Sysinternals инструмента WinObj. Рисунок 3-3 показывает этот каталог (с именем `Global ??` в WinObj).

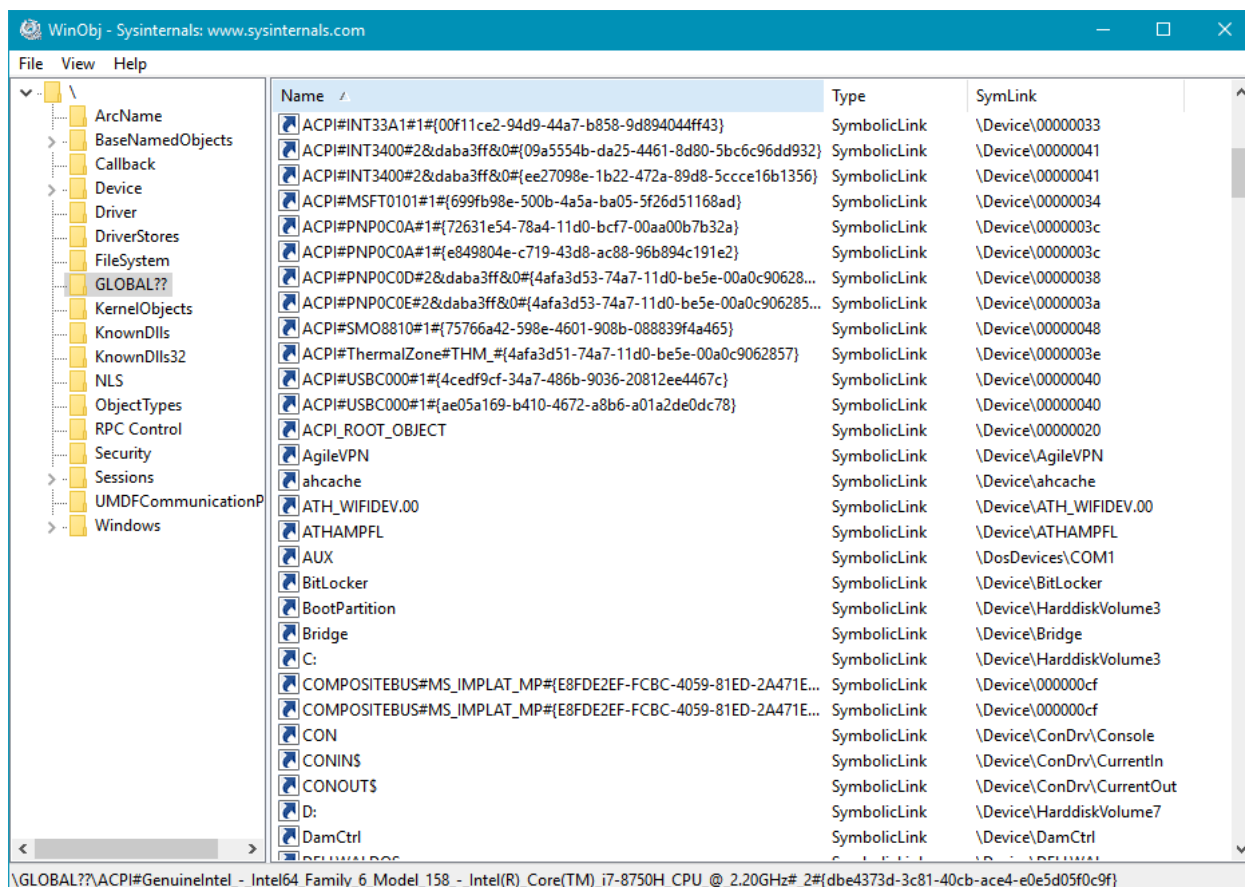


Рисунок 3-3: Символические ссылки в WinObj

Некоторые имена кажутся знакомыми, такие как C :, Aux, Con и другие. Другие записи выглядят как длинные загадочные строки, и они на самом деле генерируются системой ввода/вывода для аппаратных драйверов, которые вызывают IoRegisterDeviceInterface.

Эти типы символических ссылок бесполезны для целей этой книги.

Большинство символических ссылок в каталог указывает на внутреннее имя устройства.

Имена в этом каталоге не доступны напрямую из пользовательского режима. Но они могут быть доступным вызывающим ядрам с помощью API IoGetDeviceObjectPointer.

Каноническим примером является драйвер для Process Explorer.

Когда Process Explorer запускается с правами администратора, он устанавливает драйвер. Этот драйвер дает Process Explorer полномочия помимо тех, которые могут быть получены с помощью API пользовательского режима, даже если они работают с повышенными правами.

Например, Process Explorer в своем диалоге потока для процесса может показать полный стек вызовов потока, включая функции в режим ядра. Этот тип информации невозможно получить из пользовательского режима.

Драйвер, установленный Process Explorer, создает один объект устройства, чтобы Process Explorer мог открыть дескриптор этого устройства и сделать запросы.

Это означает, что объект устройства должен быть назван и должен иметь символическую ссылку в каталог, с именем PROCEXP152, вероятно, с указанием версии драйвера 15.2 (на момент написания этой статьи).

Рисунок 3-4 показывает эту символическую ссылку Process Explorer в WinObj.



Обратите внимание, что символическая ссылка для устройства Process Explorer указывает на \\Device\\PROCEXP152, который является внутренним именем доступным только для ядра.

Фактический вызов CreateFile, выполненный Process Explorer (или любым другим клиентом), основанный на символической ссылке, должен начинаться с \\.\.

Вот как Process Explorer может открыть дескриптор своего устройства (обратите внимание на двойную обратную косую черту):

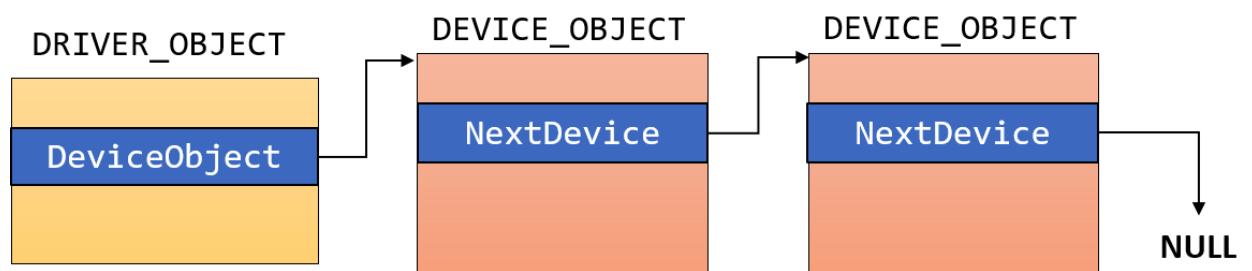
```
HANDLE hDevice = CreateFile(L"\\\\.\\PROCEXP152",  
GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);
```

Драйвер создает объект устройства с помощью функции IoCreateDevice. Эта функция выделяет и инициализирует структуру объекта устройства и возвращает его указатель вызывающей стороне.

Экземпляр объекта устройства хранится в элементе DeviceObject структуры DRIVER\_OBJECT.

Если более одного устройства, они образуют односвязный список, в котором член NextDevice из DEVICE\_OBJECT указывает на следующий объект устройства.

Обратите внимание, что объекты устройства вставляются в начало списка, поэтому первый созданный объект устройства сохраняется последним; его NextDevice указывает на NULL.



## Итоги главы

Мы рассмотрели некоторые фундаментальные структуры данных ядра и API. В следующей главе мы напишем полный драйвер и его клиент.