

## **Windows Kernel Programming. Часть 1: Внутренние компоненты Windows. Обзор**

Это первая часть перевода/пересказа/обзора (Незнаю как эту работу назвать) книги «Windows Kernel Programming Pavel Yosifovich», читаю/перевожу для себя, для начала буду выкладывать переведенные главы на форуме (<https://ru-sfera.org/forums/windows-kernel-programming.161/>), затем всё объединю в итоговый pdf оформлю как книгу.)

Возможно данный материал покажется кому-то полезным.

Также хочу отметить, что данная книга не бесплатная, если вам понравился перевод, то можете поддержать автора, купив книгу тут: ([leanpub.com/windowskernelprogramming](http://leanpub.com/windowskernelprogramming)).

**Также можно поддержать и меня (хе-хе), перевод распространяется бесплатно, так-что ставьте лайки и подписывайтесь на канал.**

Текущий материал, описывает не только внутреннее устройства ОС, но и современные подходы к проектированию и реализации драйверов, в целом такой инфы не найти в рунете, правда и интересующихся немного.)))

### **Итак начинаю перевод...**

**Таже это не совсем перевод, что-то буду объяснять как я понимаю, можно рассматривать этот материал, как пересказ что было написано в книге...**

В этой главе описаны наиболее важные понятия во внутренней работе Windows.

Некоторые темы будут описаны более подробно позже в книге.

**Убедитесь, что вы понимаете концепции, изложенные в этой главе, так как они закладывают основу для разработки драйверов.**

**В этой главе рассматриваются:**

- Процессы;
- Виртуальная память;
- Потoki;
- Системные сервисы;
- Архитектура системы;
- Handles и Objects.

Итак начнем:

## **1) Процессы**

**Процесс** - это объект исполнения и управления, представляющий работающий экземпляр программы.

Термин «процесс запускается», который используется довольно часто, является неточным.

Процессы не запускаются – процессы управляют.

**Потоки** - это то-что выполняет код и технически выполняется.

У процесса может-быть несколько потоков, т.е. поток это по сути «подпрограмма», какой-то глобальной программы (Процесса).

Поток может выполняться псевдопараллельно...

**С точки зрения высокого уровня, процесс имеет следующие части:**

- Исполняемая программа, которая содержит исходный код и данные, используемые для выполнения кода в процессе.
- Привилегированное виртуальное адресное пространство, используемое для выделения памяти для любых задач кода.
- Основной токен, который является объектом, хранит контекст безопасности процесса по умолчанию, используется потоками, выполняющими код внутри процесса.
- Приватная таблица дескрипторов для исполняемых объектов, таких как события, семафоры и файлы.
- Один или несколько потоков исполнения.

Процесс пользовательского режима создается с одним потоком (выполнение классической функции `main/WinMain`).

Процесс пользовательского режима без потоков в основном бесполезны и при нормальных обстоятельствах будут уничтожены ядром.

**Описанные элементы процесса изображены на рисунке 1-1.**

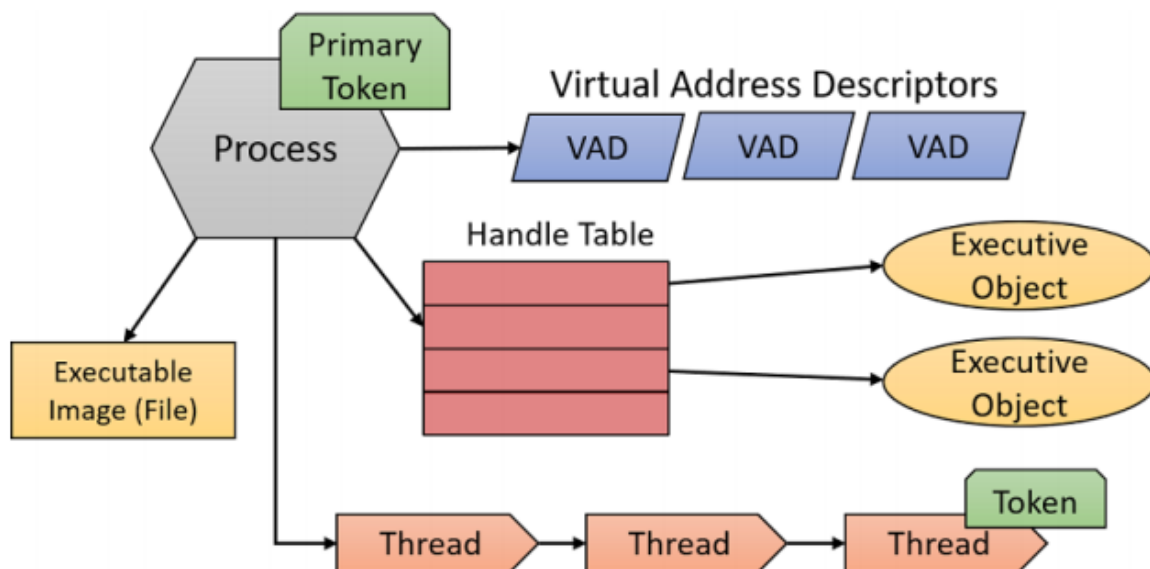


Рисунок 1-1. Элементы процесса

Процесс однозначно идентифицируется по его идентификатору процесса, который остается уникальным, пока процесс выполняется.

После того, как он уничтожен, этот же идентификатор может быть повторно использован для новых процессов.

Важно понимать, что сам исполняемый файл не является уникальным идентификатором процесса.

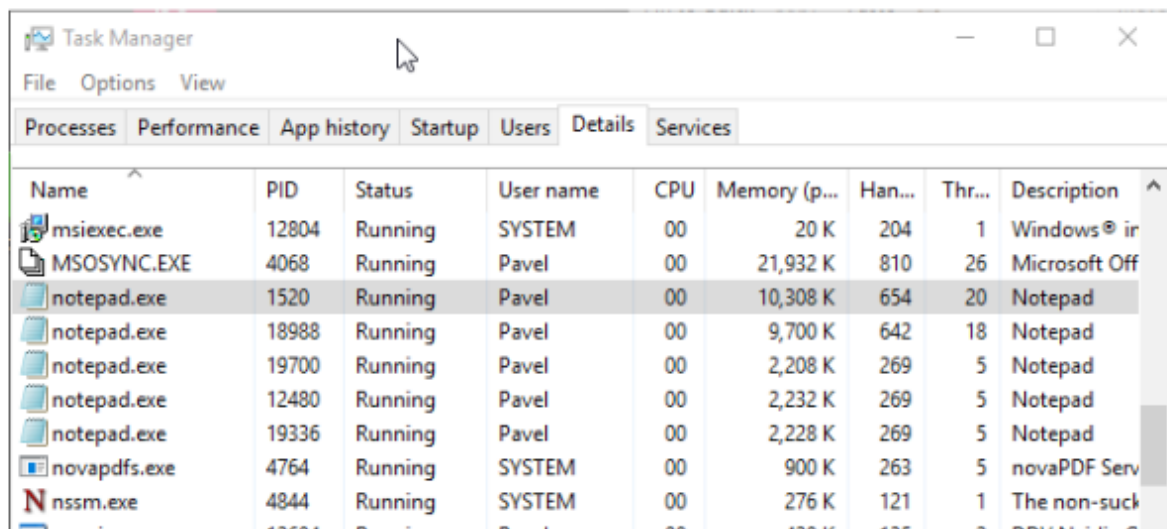
Например, может быть пять экземпляров «notepad.exe» работающих одновременно.

Каждый процесс имеет свое адресное пространство, свои собственные потоки, собственную таблицу дескрипторов, собственный уникальный идентификатор процесса и т. д.

Все эти пять процессов используют тот же файл (notepad.exe).

Рисунок 1-2 показывает снимок экрана с диспетчера задач.

На вкладке «Сведения» диспетчера задач показаны пять экземпляров Notepad.exe, каждый со своими атрибутами.



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. The window title is 'Task Manager' and it has a menu bar with 'File', 'Options', and 'View'. Below the menu bar are tabs for 'Processes', 'Performance', 'App history', 'Startup', 'Users', 'Details', and 'Services'. The 'Processes' tab is active, displaying a list of running processes with columns for Name, PID, Status, User name, CPU, Memory (private), Handles, Threads, and Description. The processes listed are: msisexec.exe (PID 12804, Running, SYSTEM, 0% CPU, 20 K Memory), MSOSYNC.EXE (PID 4068, Running, Pavel, 0% CPU, 21,932 K Memory), notepad.exe (PID 1520, Running, Pavel, 0% CPU, 10,308 K Memory), notepad.exe (PID 18988, Running, Pavel, 0% CPU, 9,700 K Memory), notepad.exe (PID 19700, Running, Pavel, 0% CPU, 2,208 K Memory), notepad.exe (PID 12480, Running, Pavel, 0% CPU, 2,232 K Memory), notepad.exe (PID 19336, Running, Pavel, 0% CPU, 2,228 K Memory), novapdfs.exe (PID 4764, Running, SYSTEM, 0% CPU, 900 K Memory), and nssm.exe (PID 4844, Running, SYSTEM, 0% CPU, 276 K Memory).

Name	PID	Status	User name	CPU	Memory (p...	Han...	Thr...	Description
msisexec.exe	12804	Running	SYSTEM	00	20 K	204	1	Windows® ir
MSOSYNC.EXE	4068	Running	Pavel	00	21,932 K	810	26	Microsoft Off
notepad.exe	1520	Running	Pavel	00	10,308 K	654	20	Notepad
notepad.exe	18988	Running	Pavel	00	9,700 K	642	18	Notepad
notepad.exe	19700	Running	Pavel	00	2,208 K	269	5	Notepad
notepad.exe	12480	Running	Pavel	00	2,232 K	269	5	Notepad
notepad.exe	19336	Running	Pavel	00	2,228 K	269	5	Notepad
novapdfs.exe	4764	Running	SYSTEM	00	900 K	263	5	novaPDF Serv
nssm.exe	4844	Running	SYSTEM	00	276 K	121	1	The non-suck

Рисунок 1-2. Снимок экрана с диспетчера задач

## 2) Виртуальная память

Каждый процесс имеет свое собственное виртуальное, приватное, линейное адресное пространство.

Это адресное пространство пустое (или близко к пустому, поскольку исполняемый образ и NtDll.Dll отображаются первыми, а затем при необходимости подгружаются DLL).

Как только начинается выполнение основного (первого) потока, при необходимости загружаются больше DLL и т. д.

Это адресное пространство является приватным, что означает другие процессы не могут получить к нему доступ напрямую.

*Короче, если не понятно (инглиш не мой канек), можно проще объяснить, эта память выделяется в процессе работы программы, изначально её как-бы и нет.)))*

Диапазон адресного пространства начинается с нуля (хотя технически первые 64 КБ не может быть выделено или использовано каким-либо образом), и идет до максимума, который зависит от «битности» процесса (32 или 64 бит) и «битности» операционной системы следующим образом:

- Для 32-разрядных процессов в 32-разрядных системах Windows размер адресного пространства процесса составляет 2 ГБ.
- Для 32-разрядных процессов в 64-разрядных системах Windows, в которых используется параметр увеличения адресного пространства пользователя. (Флаг LARGEADDRESSAWARE в заголовке Portable Executable), размер

адресного пространства этого процесса может достигать 3 ГБ (в зависимости от точной настройки).

Чтобы получить расширенное адресное пространство, исполняемый файл, из которого был создан процесс, должен быть помечен флагом компоновщика LARGEADDRESSAWARE в его заголовке.

Если это не так, он все равно будет ограничен 2 ГБ.

- Для 64-разрядных процессов (естественно, в 64-разрядной системе Windows) размер адресного пространства составляет 8 ТБ (Windows 8 и более ранние версии) или 128 ТБ (Windows 8.1 и более поздние версии).
- Для 32-разрядных процессов в 64-разрядной системе Windows размер адресного пространства составляет 4 ГБ, если исполняемый файл был собран с флагом LARGEADDRESSAWARE. В противном случае размер остается на уровне 2 ГБ.

#### **Примечание:**

*Требование флага LARGEADDRESSAWARE связано с тем, что адрес объемом 2 ГБ требует только 31 бита, оставляя старший значащий бит (MSB) свободным для использования приложением.*

*Указание этого флага указывает на то, что программа не использует бит 31 ни для чего, и поэтому установка этого бита в 1 (что произошло бы для адресов размером более 2 ГБ) не является проблемой.*

Каждый процесс имеет свое собственное адресное пространство, что делает любой адрес процесса относительным, а не абсолютным.

Например, при попытке определить, что лежит в адресе 0x20000, самого адреса недостаточно. Процесс, к которому относится этот адрес, может быть специфичен.

Сама память называется виртуальной, что означает наличие косвенной связи между адресным диапазоном и точным местоположением, где он находится в физической памяти (RAM).

Буфер внутри процесса может быть сопоставлен с физической памятью или временно находиться в файле (например, в файле подкачки).

Термин виртуальный относится к тому факту, что с точки зрения исполнения, нет необходимости знать, находится ли память в ОЗУ или нет.

Если память действительно отображается в ОЗУ, процессор получит доступ к данным напрямую.

Если нет, ЦП вызовет исключение ошибки страницы, которое вызовет обработчик ошибок страницы диспетчера памяти для извлечения данных из соответствующего файла и копирования их в оперативную память.

На рисунке 1-3 показано это сопоставление виртуальной и физической памяти для двух процессов.

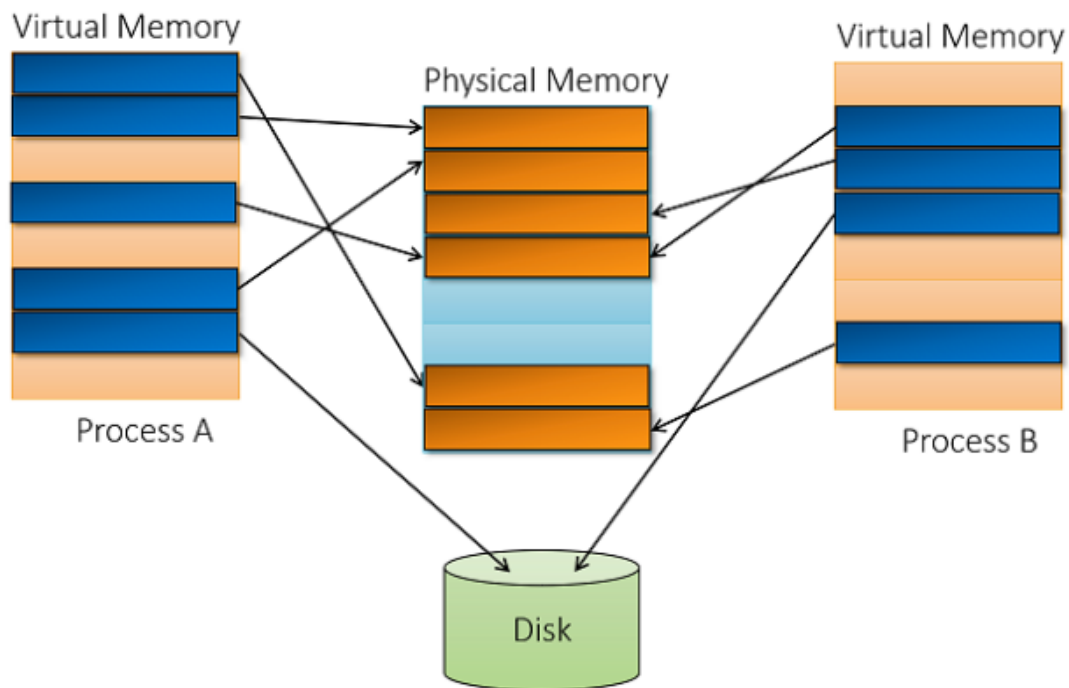


Рисунок 1-3. Сопоставление виртуальной и физической памяти для двух процессов.

Единица управления памятью называется страницей.

Каждый атрибут, связанный с памятью, всегда находится на детализации страницы, например ее защита.

Размер страницы определяется типом процессора и на некоторых процессорах, могут быть конфигурируемыми.

Обычный (иногда называемый маленьким) размер страницы составляет 4 КБ на всех поддерживаемых Windows архитектурах.

Помимо нормального (небольшого) размера страницы, Windows также поддерживает большие страницы.

Большой размер страницы составляет 2 МБ (x86 / x64 / ARM64) и 4 МБ (ARM).

Это основано на использовании записи каталога страниц Page Directory Entry (PDE) для отображения большой страницы без использования таблицы страниц.

Это приводит к более быстрой трансляции, но лучше использовать Translation Lookaside Buffer (TLB) - кеш недавно транслированных страниц поддерживаемых процессором.

В случае большой страницы одна запись TLB может отображать значительно больше памяти, чем маленькая страница.

Недостатком больших страниц является необходимость иметь непрерывную память в ОЗУ, что может-быть плохо, если память ограничена или сильно фрагментирована.

Кроме того, большие страницы не всегда являются страницами и должны быть защищены только доступом для чтения /записи.

Огромные страницы размером 1 ГБ поддерживаются на Windows 10 и Server 2016 и более поздних версиях.

Они используются автоматически, если выделение памяти занимает не менее 1 ГБ, и эта страница может быть расположена как непрерывная в ОЗУ.

### **3)Состояние страниц виртуальной памяти**

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

- Free — В этой странице ничего нет. Любая попытка доступа к этой страницы вызовет исключение нарушения прав доступа. Большинство страниц во вновь созданном процессе являются свободными.
- Committed - Выделенная страница, к которой можно успешно получить доступ в соответствии атрибутами защиты (например, запись на страницу только для чтения вызывает нарушение прав доступа).

Выделенные страницы обычно отображаются в ОЗУ или в файле (например, файл страницы).

- Reserved - страница не выделена, но диапазон адресов зарезервирован для возможного выделения в будущем. С точки зрения процессора, это то же самое, что и свободная страница - любая попытка доступа вызовет исключение нарушения доступа.

Тем не менее, новые попытки выделения с использованием VirtualAlloc (или NtAllocateVirtualMemory), которая не указывает конкретный адрес, не будет выделяться в зарезервированном регионе.

Классический пример использования зарезервированной памяти описана далее в этой главе в разделе «Стеки потоков».

### **4)Системная память**

Нижняя часть адресного пространства предназначена для использования процессами.

Пока выполняется определенный поток, его адресное пространство связанного процесса видно от нулевого адреса до верхнего предела, как описано в предыдущем разделе.

Операционная система, однако, также должна находиться где-то - и это где-то это верхний диапазон адресов, который поддерживается в системе следующим образом:

- В 32-разрядных системах, работающих без настройки увеличения виртуального адресного пространства пользователя, операционная система находится в верхних 2 ГБ виртуального адресного пространства, от адреса 0x80000000 до 0xFFFFFFFF.



- В 32-разрядных системах, настроенных с настройкой увеличения виртуального адресного пространства пользователя, операционная система находится в левом адресном пространстве.

Например, если система настроена с 3 ГБ, адресное пространство пользователя на процесс (максимальное), ОС занимает верхний 1 ГБ (от адреса, т. е. от 0xC0000000 до 0xFFFFFFFF).

Сущность, которая в основном страдает от этого сокращения адресного пространства, это кеш файловой системы.

- В 64-разрядных системах в Windows 8, Server 2012 и более ранних версиях ОС требуется более 8 ТБ виртуального адресного пространства.
- В 64-разрядных системах под управлением Windows 8.1, Server 2012 R2 и более поздних версий ОС занимает верхние 128 ТБ под виртуальное адресное пространство.

Системное пространство не относится к процессу - в конце концов, это та же «система», то же ядро, также и драйверы, которые обслуживают каждый процесс в системе (за исключением некоторой системной памяти, которая включена для каждой сессии, но это не важно для этого обсуждения).

Отсюда следует, что любой адрес в системе является абсолютным, а не относительным, поскольку он «выглядит» одинаково в любом контексте процесса.

Конечно, фактический доступ из пользовательского режима в системное пространство приводит к исключению нарушения прав доступа.

В системном пространстве находятся само ядро, уровень абстрагирования оборудования (HAL) и драйверы ядра.

Таким образом, драйверы ядра автоматически защищены от прямого доступа в пользовательском режиме.

Это также означает, что они оказывают общесистемное влияние.

Например, если драйвер ядра вызывает утечку памяти, то эта память не будет освобождена даже после выгрузки драйвера.

У процессов пользовательского режима, память будет всегда очищена, после его завершения.

Ядро отвечает за закрытие и освобождение всего приватного для мертвого процесса (все дескрипторы закрыты и вся приватная память освобождена).

## 5)Потоки

Фактические сущности, которые выполняют код, являются потоками. Поток содержится в процессе, используя ресурсы, предоставляемые процессом для выполнения работы (например, виртуальная память и дескрипторы объектов ядра).

**Самая важная информация, которой владеет поток, следующая:**

- Текущий режим доступа, пользователь или ядро.
- Контекст выполнения, включая регистры процессора и состояние выполнения.
- Один или два стека, используемые для распределения локальных переменных и управления вызовами.
- Массив локального хранилища потоков (TLS), который обеспечивает способ хранения личных данных потоков с семантикой унифицированного доступа.
- Базовый приоритет и текущий (динамический) приоритет.
- Привязка к процессору, указывающая, на каких процессорах разрешено запускать поток.

Наиболее распространенные состояния, в которых может находиться поток:

- Выполняется - в настоящее время выполняется код на (логическом) процессоре.
- Готов - ожидание запланированного выполнения, потому что все соответствующие процессоры заняты или недоступен.
- Ожидание - ожидание какого-либо события перед продолжением. Как только событие происходит, поток переходит в состояние готовности.

Рисунок 1-4 показывает диаграмму состояний.

Цифры в скобках указывают состояние, это можно увидеть с помощью таких инструментов, как Performance Monitor.

Обратите внимание, что состояние готовности имеет одноуровневое состояние, называемое Deferred Ready, которое аналогично и необходимо, чтобы минимизировать некоторые внутренние блокировки.

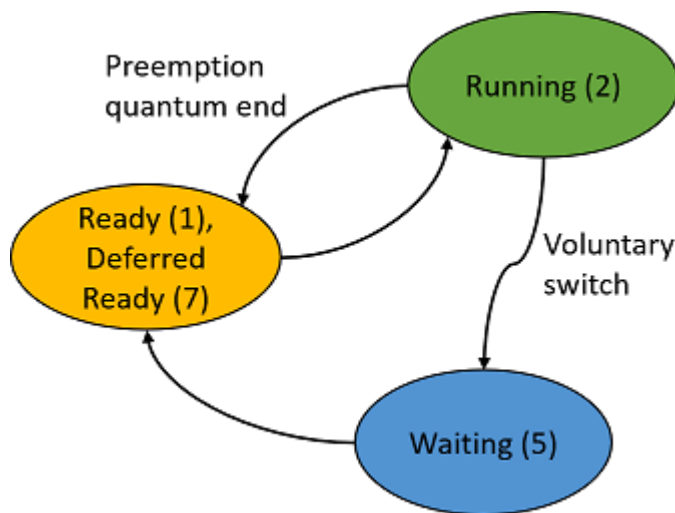


Рисунок 1-4. Диаграмма состояний потоков

## 6)Стеки потоков

Каждый поток имеет стек, который он использует при выполнении, стек используется для локальных переменных, передачи параметров в функции (в некоторых случаях), в стеке также хранятся адреса возврата до вызова функций.

Поток имеет по крайней мере один стек, находящийся в системном пространстве (ядре), и он довольно мал (по умолчанию 12 КБ для 32-разрядные системы и 24 КБ в 64-разрядных системах).

Поток пользовательского режима имеет второй стек в своем процессе.

Диапазон адресов пространства пользователя значительно больше (по умолчанию может увеличиваться до 1 МБ).

Пример три потока пользовательского режима и их стеки показаны на рисунке 1-5.

На рисунке потоки 1 и 2 находятся в процессе А и поток 3 находится в процессе В.

Стек ядра всегда находится в оперативной памяти, пока поток находится в состоянии «Выполнено» или «Готов».

Стек режима пользователя, может быть выгружен, как и любой пользовательский режим памяти.

Стек режима пользователя обрабатывается иначе, чем стек режима ядра, с точки зрения его размера.

Он начинается с небольшого объема выделенной памяти (может быть как одна страница), остальное адресное пространство стека, является зарезервированной памятью, что означает, что оно никоим образом не выделено.

Идея состоит в том чтобы иметь возможность увеличивать стек в случае, если код потока использует больше места в стеке.

Чтобы сделать это, следующая страница (иногда более одной) сразу после выделенной части помечается специальной защитой PAGE\_GUARD - эта страница защищена.

Если потоку требуется больше места в стеке он будет писать на страницу защиты, которая выдаст исключение, которое обрабатывается менеджером памяти. Затем диспетчер памяти снимает защитную метку, фиксирует страницу и отмечает следующую страницу в качестве защищенной страницы. Таким образом, стек увеличивается по мере необходимости.

Рисунок 1-6 показывает, как выглядит стек потоков пользовательского режима.

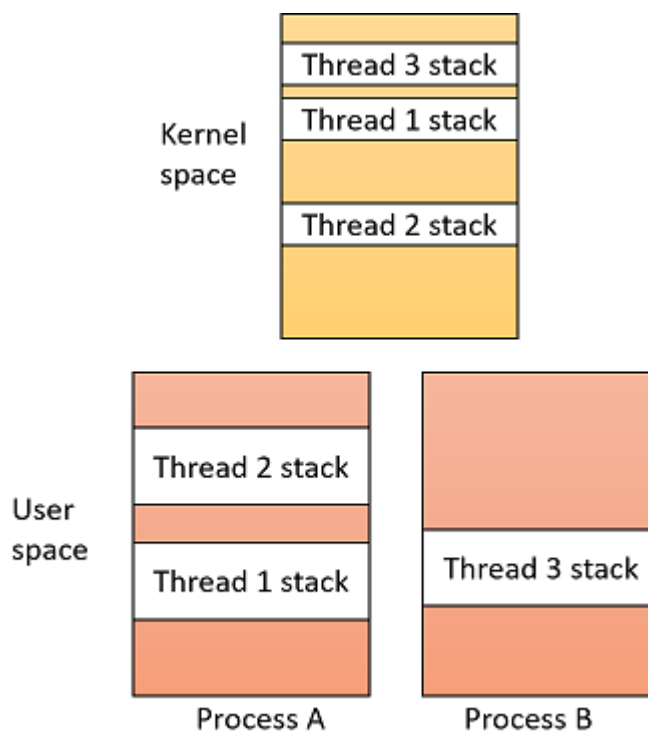


Рисунок 1-5.Стеки потоков

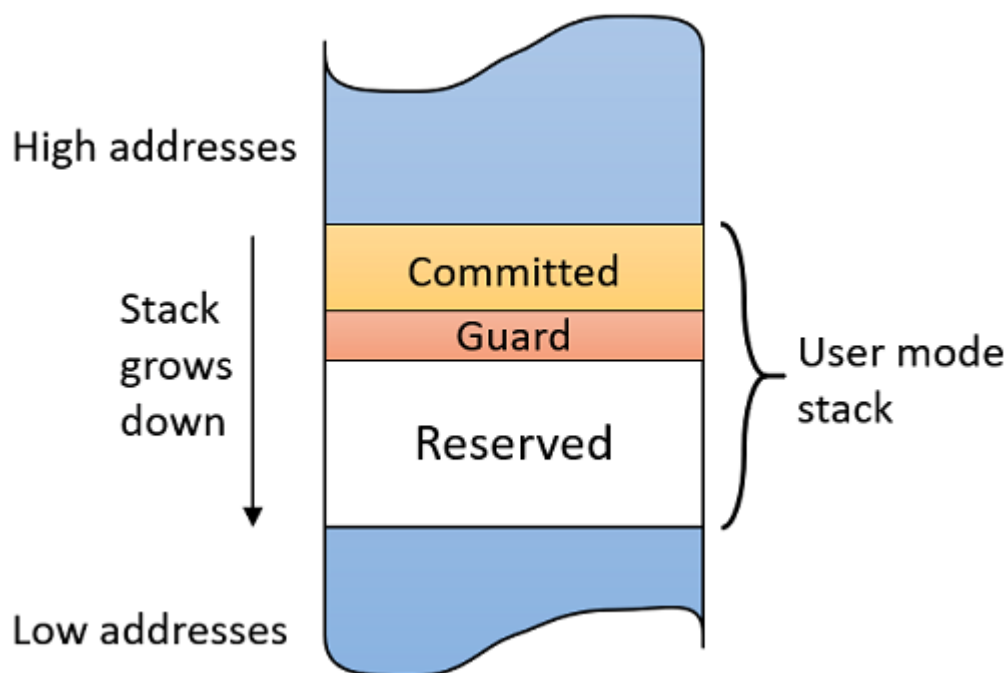


Рисунок 1-6.Выделение памяти в стеке потоков

- У исполняемого образа есть выделенный стек и зарезервированные значения в его Portable Executable (PE) заголовке. Они принимаются как значения по умолчанию, если поток не указывает альтернативные значения.
- Когда поток создается с помощью `CreateThread` (и аналогичных функций), вызывающая сторона может указать его требуемый размер стека, либо predetermined фиксированный размер, либо зарезервированный размер (но не оба).

**Примечание:**

*Любопытно, что функции `CreateThread` и `CreateRemoteThread (Ex)` позволяют указав одно значение для размера стека и может быть зафиксированным или зарезервированным размером, но не оба.*

*Нативная (недокументированная) функция `NtCreateThreadEx` позволяет указывать оба значения.*

## 7) Системные сервисы (Системные вызовы)

Приложения должны выполнять различные операции, которые не являются чисто вычислительными, такие как выделение памяти, открытие файлов, создание потоков и т. д.

Эти операции могут выполняться кодом, работающим только в режиме ядра.

Итак, как код пользовательского режима сможет выполнить такие операции?

Давайте рассмотрим классический пример: пользователь, запускающий процесс «Блокнот», использует меню «Файл» для запроса на открытие файла.

Код Блокнота отвечает, вызывая документированный API Windows CreateFile.

CreateFile задокументирован как реализованный в библиотеке kernel32.Dll, одной из Windows DLL подсистемы.

Эта функция по-прежнему работает в пользовательском режиме, поэтому нет возможности напрямую открыть файл.

После некоторой проверки ошибок он вызывает NtCreateFile, функцию, реализованную в NTDLL.dll, это основополагающая DLL, которая реализует API, известный как «Native API», и фактически является самым низким слоем кода, который все еще находится в режиме пользователя.

**Это официально недокументированный API** - это API, который осуществляет переход в режим ядра.

Перед фактическим переходом ставится число, называемое системным сервисным номером, в регистр процессора (EAX на архитектурах Intel / AMD).

Затем выдается специальная инструкция процессору (syscall на x64 или sysenter на x86), которая осуществляет фактический переход в режим ядра.

Переход к предварительно определенной подпрограмме, вызывается диспетчером системных служб.

Диспетчер системных служб, в свою очередь, использует значение в регистре EAX в качестве индекса в Service Dispatch Table (SSDT).

Используя эту таблицу, код переходит на нужный системный сервис (системный вызов).

В нашем примере с Блокнотом, запись SSDT будет указывать на функцию NtCreateFile менеджера ввода-вывода.

Обратите внимание, что функция имеет то же имя, что и в NTDLL.dll, и фактически имеет те же самые аргументы.

Как только системная служба завершена, поток возвращается в пользовательский режим для выполнения инструкции следующей за `sysenter / syscall`. Эта последовательность событий изображена на рисунке 1-7.

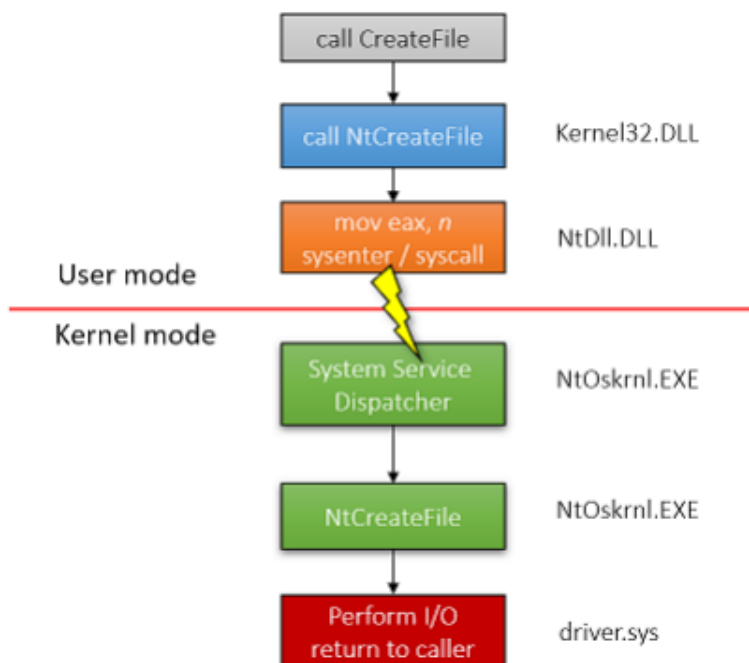


Рисунок 1-7. Процесс выполнения системного вызова в системе

## 8)Общая системная архитектура системы

Рисунок 1-8 показывает общую архитектуру Windows, состоящую из компонентов пользовательского режима и режима ядра.

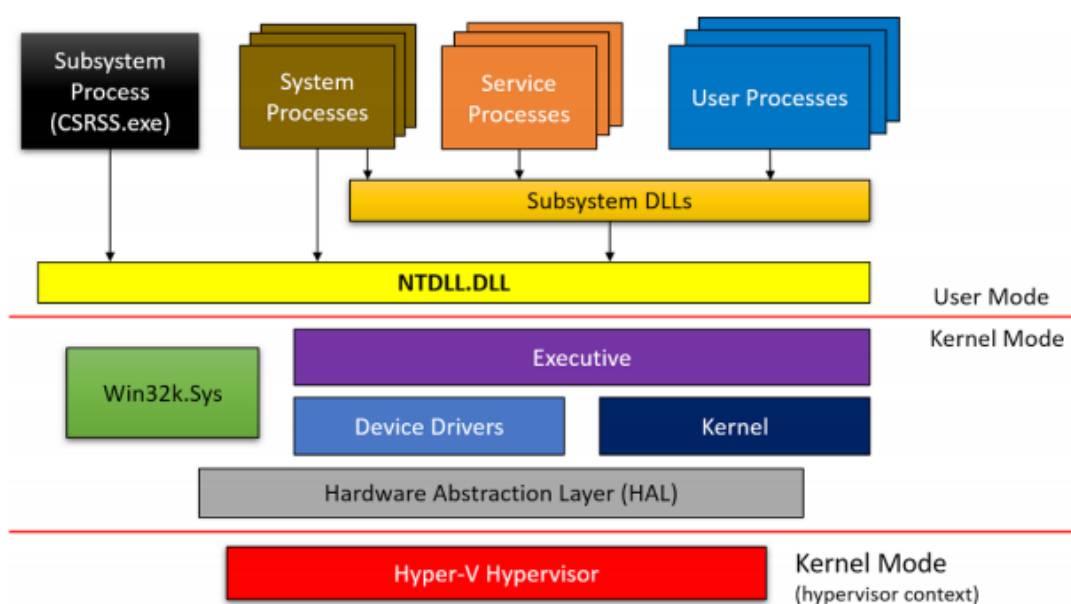


Рисунок 1-8. Общая архитектура Windows

**Вот краткое изложение названных блоков, показанных на рисунке 1-8:**

- **Пользовательские процессы:**

Это обычные процессы, основанные на отображенных файлах, выполняющихся в системе, например, экземпляры: Notepad.exe, cmd.exe, explorer.exe и так далее.

- **Подсистема DLL:**

DLL подсистема - это библиотеки динамических ссылок (DLL), которые реализуют API подсистемы.

Технически, начиная с Windows 8.1 существует только одна подсистема - подсистема Windows.

DLL подсистемы Windows включают в себя известные файлы, такие как kernel32.dll, user32.dll, gdi32.dll, advapi32.dll, combase.dll и много других.

К ним относится в основном официально документированный API Windows.

*Можно проще, в этих библиотеках находятся API Windows, которые можно использовать в своих проектах.)*

- **NTDLL.DLL:**

Общесистемная DLL, реализующая собственный API Windows. Это самый низкий уровень кода который все еще находится в режиме пользователя. Его самая важная роль - сделать переход в режим ядра.

NTDLL также реализует диспетчер кучи, загрузчик изображений и некоторую часть пула потоков пользовательского режима.

- **Сервисные процессы:**

Сервисные процессы - это обычные процессы Windows, которые взаимодействуют с Service Control Manager (SCM, реализован в services.exe) и позволяет контролировать время жизни сервисов.

SCM может запускать, останавливать, приостанавливать, возобновлять и отправлять другие сообщения службам.

Сервисы обычно выполняются под одной из специальных учетных записей Windows - локальной системы, сетевой службы или локальной службы.

- **Исполнение (Executive):**

Исполнение - это верхний уровень NtOskrnl.exe («ядро»). Он содержит большую часть кода, который в режиме ядра.

В основном это различные «менеджеры»: диспетчер объектов, диспетчер памяти, диспетчер ввода-вывода, Plug & Play Manager, Power Manager, Configuration Manager и т. д.

- **Ядро:**

Уровень ядра реализует наиболее фундаментальные и чувствительные ко



времени части ОС в режиме ядра.

Ядро включает в себя планирование потоков, обработку прерываний и исключений, а также реализацию различных примитивов ядра, таких как мьютекс и семафор.

Часть кода ядра написана на машинном языке, специфичном для процессора, для эффективности и для получения прямого доступа к регистрам, специфичному для процессора.

- **Драйверы устройств:**

Драйверы устройств - это загружаемые модули ядра. Их код выполняется в режиме ядра. Эта книга посвящена написанию определенных типов драйверов ядра.

- **Win32k.sys:**

«Компонент режима ядра подсистемы Windows». По сути это модуль ядра (драйвер), который обрабатывает часть пользовательского интерфейса Windows и классические API-интерфейсы графического устройства (GDI). Это означает, что все оконные операции (CreateWindowEx, GetMessage, PostMessage и т.д.) обрабатываются этим компонентом.

- **Уровень аппаратной абстракции (HAL):**

HAL - это уровень абстракции над оборудованием, ближайшим к процессору. Это позволяет драйверам устройств использовать API, которые не требуют подробных и конкретных знаний таких вещей, как контроллер прерываний или контроллер DMA. Естественно, этот слой в основном полезен для драйверов устройств, написанных для обработки аппаратных устройств.

- **Системные процессы:**

Системные процессы - это общий термин, используемый для описания процессов, которые обычно «просто существуют».

Данные процессы нужны для совершения некоторых системных действий, например вход пользователя в систему (Winlogon.exe) .

Тем не менее, это критичные процессы, завершение некоторых из них может привести к сбою системы, или неправильной её работы.

Некоторые системные процессы являются нативными процессами, это означает, что они используют только собственный API (API, реализованный NTDLL).

Пример системных процессов: Smss.exe, Lsass.exe, Winlogon.exe, Services.exe и другие.

- **Подсистема Процесс:**

Процесс подсистемы Windows, выполняющий образ Csrss.exe, может

рассматриваться как помощник ядра для управления процессами, работающими в системе Windows.

Это критический процесс, это означает, что если он будет остановлен, система потерпит крах.)

Обычно есть один экземпляр Csrss.exe для сеанса, поэтому в стандартной системе существует два экземпляра - один для сеанса ядра (0) и один для сеанса пользователя, вошедшего в систему (обычно 1).

#### • Hyper-V Hypervisor

Гипервизор Hyper-V существует в Windows 10 и сервере 2016 (и более поздних версиях).

Поддержка виртуализации на основе безопасности (VBS).

VBS обеспечивает дополнительный слой уровня 0, фактически машина является виртуальной машиной, управляемой Hyper-V.

Для получения дополнительной информации, ознакомьтесь с книгой «**Windows Internals**».

#### **Примечание:**

*В Windows 10 версии 1607 была представлена подсистема Windows для Linux (WSL).*

*Хотя это может выглядеть как еще одна подсистема, например, поддерживаемые старые подсистемы POSIX и OS / 2 Windows, но это совсем не так.*

*Старые подсистемы могли выполнять POSIX и приложения OS / 2, если программы были скомпилированы на компиляторе Windows.*

*WSL, с другой стороны, не имеет такого требования. Существующие исполняемые файлы из Linux (хранящиеся в формате ELF) могут быть запущены как есть на винде, без перекомпиляции.*

*Чтобы это сделать, был создан новый тип процесса - процесс Pico вместе с провайдером Pico.*

*Вкратце, процесс Pico - это пустое адресное пространство (минимальный процесс), который используется для процессов WSL, где каждый системный вызов (системный вызов Linux) должен быть перехвачен и переведен в эквивалент системных вызовов Windows, используя для этого провайдер Pico (драйвер устройства).*

*На компьютере с Windows установлен настоящий Linux (часть пользовательского режима).*

## **9) Handles and Objects**

Ядро Windows предоставляет различные типы объектов для использования процессами пользовательского режима, ядром и драйверами режима ядра.

Экземплярами этих типов являются структуры данных в системном пространстве, созданные диспетчером объектов (часть исполнения) по запросу пользователя или кода режима ядра.

Объекты считаются ссылками - только когда последняя ссылка на объект будет освобождена, этот объект может-быть уничтоженным и освобожденным в памяти.

Поскольку эти экземпляры объектов находятся в системном пространстве, они не могут быть доступны напрямую в режиме пользователя.

Пользовательский режим должен использовать механизм косвенного доступа, известный как дескрипторы.

Дескриптор является указателем на запись в таблице, поддерживаемая процессом по каждому процессу, которая логически указывает на объект ядра проживающий в системном пространстве.

Существуют различные функции Create \* и Open \* для создания/открытия объектов и восстанавливающие обратные маркеры для этих объектов.

Например, функция пользовательского режима CreateMutex позволяет делать создание или открытие мьютекса (в зависимости от того, существует-ли объект).

В случае успеха функция возвращает дескриптор объекта.

Возвращаемое значение ноль означает недопустимый дескриптор (и сбой вызова функции). Функция OpenMutex, с другой стороны, пытается открыть дескриптор для именованного мьютекса. Если мьютекс с таким именем не существует, функция завершается ошибкой и возвращает ноль (0).

Код ядра (и драйвера) может использовать либо дескриптор, либо прямой указатель на объект.

В некоторых случаях дескриптор, полученный пользовательским режимом для драйвера должен быть превращен в указатель с помощью функции ObReferenceObjectByHandle.

Мы обсудим эти подробности в следующей главе.

### ***Примечание:***

*Большинство функций возвращают ноль (ноль) при сбое, но некоторые нет. В частности, CreateFile возвращает INVALID\_HANDLE\_VALUE (-1), если произошла ошибка.*

Значения дескриптора кратны 4, где первый действительный дескриптор равен 4.

Ноль никогда не является допустимым значением дескриптора.

Код режима ядра может использовать дескрипторы при создании /открытии объектов, но они также могут использовать прямой указатель на объекты ядра.

Обычно это делается, когда этого требует определенный API. Код ядра может получить указатель на объект с заданным допустимым дескриптором, используя функцию ObReferenceObjectByHandle.

В случае успеха счетчик ссылок на объект увеличивается.

Если пользовательская программа (клиент) решила закрыть дескриптор, то вызывается функция ObDereferenceObject, которая уменьшает счетчик ссылок.

Если код ядра пропустит этот вызов, то произойдет утечка ресурсов, которая будет решена только при следующей перезагрузки системы.

Как только объект больше не нужен, его клиент должен закрыть дескриптор (если дескриптор был использован для доступа к объекту) или разыменования объекта (если клиент ядра использует указатель).

После этого, код должен считать свой дескриптор/указатель недействительным. Менеджер объектов уничтожит объект, если его счетчик ссылок достигает нуля.

Каждый объект указывает на тип объекта, который содержит информацию о самом типе

Они также отображаются как экспортированные глобальные переменные ядра, некоторые из которых определены в заголовках ядра и на самом деле полезны в определенных случаях, как мы увидим в последующих главах.

### **11)Имена объектов**

Некоторые типы объектов могут иметь имена. Эти имена могут быть использованы для открытия объектов по имени с подходящей функцией Open.

Обратите внимание, что не все объекты имеют имена; например, процессы и потоки у них нет имен - у них есть идентификаторы.

Вот почему функции `OpenProcess` и `OpenThread` требуют идентификатор процесса/потока (число), а не строковое имя.

Еще один интересный случай - объект, который не имеет имени, является файлом.

Имя файла не является именем объекта – это разные понятия.

Из кода пользовательского режима вызов функции `Create` с именем создает объект с этим именем, если объект с таким именем не существует, но если он существует, он просто открывает существующий объект.

В этом случае вызов `GetLastError` возвращает `ERROR_ALREADY_EXISTS`, указывая, что это не новый объект и возвращенный дескриптор является еще одним дескриптором существующего объекта.

Имя, предоставленное функции `Create`, на самом деле не является окончательным именем объекта.

Это предварительно добавленная строка:  
*with \Sessions \x \BaseNamedObjects \* где:

*x* - идентификатор сеанса вызывающего абонента. Если сеанс равен нулю, к имени добавляется *\BaseNamedObjects \*.

Если вызывающая сторона работает в `AppContainer` (обычно это процесс универсальной платформы `Windows`), тогда предварительно добавленная строка является более сложной и состоит из уникального `SID AppContainer`:

*\Sessions \x \AppContainerNamedObjects \{AppContainerSID}*.

Все вышеизложенное означает, что имена объектов являются относительными к сеансу.

Если объект должен быть разделен между сеансами, он может быть создан в сеансе 0, добавление имени объекта к `Global \`.

Например, создание мьютекса с помощью функции `CreateMutex` с именем `Global\MyMutex` создаст ее в `\BaseNamedObjects`.

Обратите внимание, что значения `AppContainersHandle` кратны 4, где первый действительный дескриптор равен 4.

Ноль никогда не является допустимым значением дескриптора.

Обратите внимание, что AppContainers не могут использовать пространство имен объекта сеанса 0.

Эту иерархию можно просмотреть с помощью инструмента Sysinternals WinObj (запуск с правами администратора), как показано на рисунке 1-9.

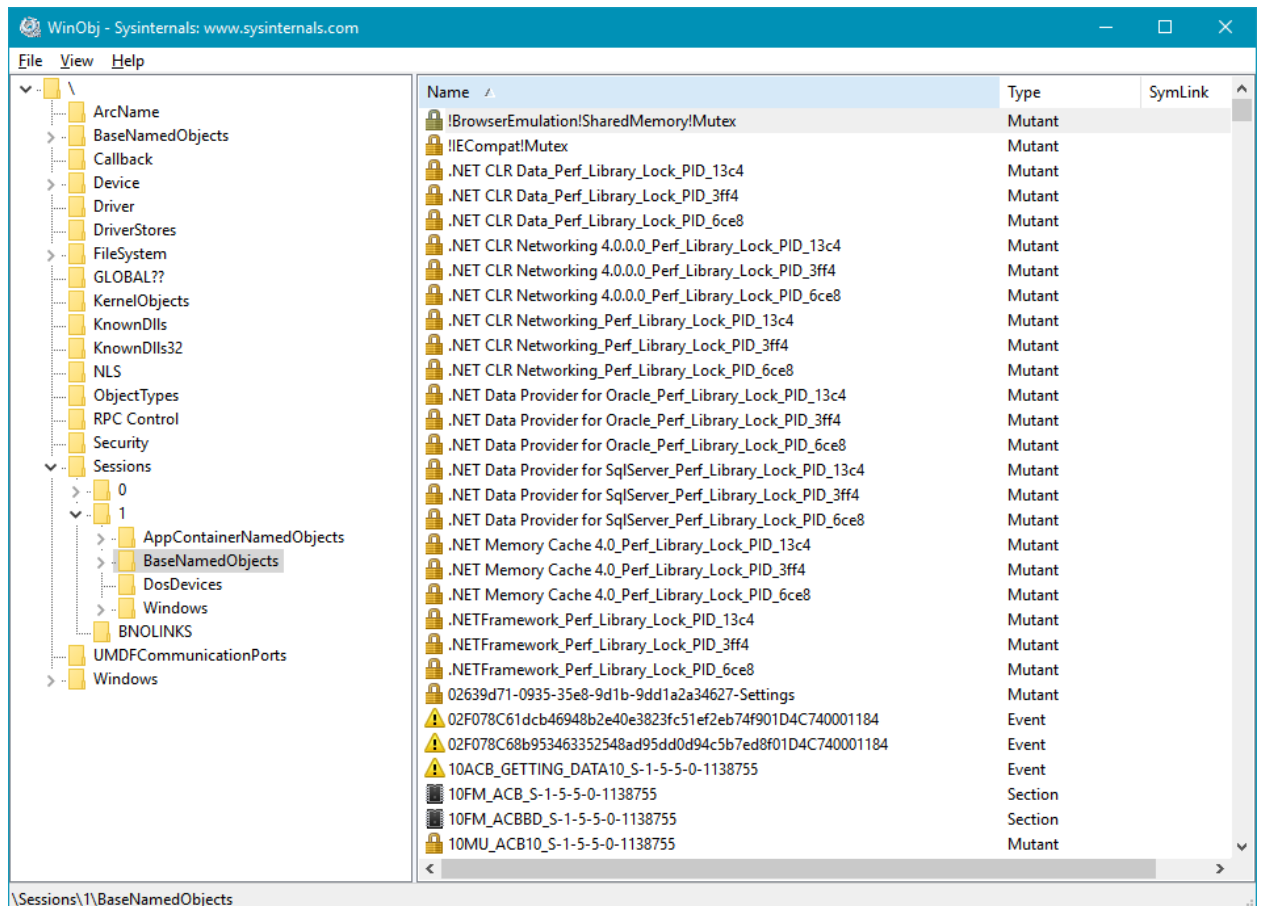


Рисунок 1-9. Иерархия имен объектов



Показанное на рисунке 1-9, является пространством имен менеджера объектов, состоящим из иерархии именованных объектов.

Вся эта структура хранится в памяти и управляется диспетчером объектов по мере необходимости. Обратите внимание, что неназванные объекты не являются частью этой структуры, то есть объекты, видимые в WinObj, включают не все существующие объекты, а все объекты, которые были созданы с именем.

Каждый процесс имеет собственную таблицу дескрипторов объектов ядра (именованных или нет), которые можно просмотреть с помощью инструментов Process Explorer и/или Handles Sysinternals.

Снимок экрана процесса «Проводник», показывающий маркеры, показан на рисунке 1-10.

Столбцы по умолчанию, показывают «handles», «тип объекта» и «имя».

Тем не менее, есть и другие доступные столбцы, как показано на рисунке 1-10.

Process	PID	CPU	Private Bytes	Working Set	User Name	Session	Protection	Priority	Handles	Start Time	Threads
dmv.exe	22992	0.36	248,332 K	185,300 K	Window Manager...	2		13	1,684	07:09:32 26-Mar-19	15
EpicGamesLauncher.exe	9416	0.22	275,084 K	25,748 K	VOYAGER\ Pavel	2		8	196,016	23:09:41 28-Mar-19	92
test_of.exe	5308		1,616 K	3,312 K	NT AUTHORITY\...	0		13	109	18:12:11 25-Mar-19	3
explorer.exe	15524	0.12	285,112 K	192,584 K	VOYAGER\ Pavel	2		8	9,501	07:51:56 27-Mar-19	144
FileCoAuth.exe	34468		10,228 K	17,732 K	VOYAGER\ Pavel	2		8	383	09:25:24 30-Mar-19	4
irefox.exe	39212	0.26	441,644 K	442,792 K	VOYAGER\ Pavel	2		8	3,092	19:22:04 28-Mar-19	68
irefox.exe	44540	0.16	227,704 K	252,988 K	VOYAGER\ Pavel	2		8	844	19:22:05 28-Mar-19	10
irefox.exe	45616	0.04	373,588 K	316,698 K	VOYAGER\ Pavel	2		8	1,411	10:33:06 28-Mar-19	27

Handle	Type	Name	Object Address	Access	Decoded Access
0x34	Directory	\KnownDlls	0xFFFFD00A8BDCAE50	0x00000003	QUERY   TRVERSE
0x40	File	C:\Windows	0xFFFFB800294B1270	0x00100020	SYNCHRONIZE   EXECUTE
0x70	Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0xFFFFD00A8B4E1320	0x00000001	QUERY_VALUE
0x84	File	C:\Windows\System32\en-US\winmm.dll.mui	0xFFFFB800205DE820	0x00120089	READ_CONTROL   SYNCHRONIZE   FILE_GENERIC_READ
0x88	Key	HKLM	0xFFFFD00A8B4E1560	0x00020019	READ_CONTROL   KEY_READ
0x98	Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0xFFFFD00A8B4E2880	0x00020019	READ_CONTROL   KEY_READ
0xA4	File	\Device\NCG	0xFFFFB8001C59F660	0x00100001	SYNCHRONIZE   READ_DATA
0xA8	Key	HKLM\SOFTWARE\Microsoft\OLE	0xFFFFD00A8B4E1C20	0x00020019	READ_CONTROL   KEY_READ
0xB0	Key	HKCU\Software\Classes\Local Settings\Software\Microsoft	0xFFFFD00A8B4E17A0	0x00020019	READ_CONTROL   KEY_READ
0xB4	Key	HKCU\Software\Classes\Local Settings	0xFFFFD00A8B4E19E0	0x00020019	READ_CONTROL   KEY_READ
0xE4	Directory	\Sessions\2\BaseNamedObjects	0xFFFFD00A348C840	0x0000000F	QUERY   TRVERSE   CREATE_OBJECT   CREATE_SUBDIRECTORY
0x110	Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File ...	0xFFFFD00A8B4E1B00	0x00000009	QUERY_VALUE   ENUMERATE_SUB_KEYS
0x11C	Window Station	\Sessions\2\Windows\Window Stations\WinSta0	0xFFFFB80017C496C0	0x000F037F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   WINSTA_ALL_ACCESS
0x120	Desktop	\Default	0xFFFFB800A038C40	0x000F01FF	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   ENUMERATE   READOBJ
0x124	Window Station	\Sessions\2\Windows\Window Stations\WinSta0	0xFFFFB80017C496C0	0x000F037F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   WINSTA_ALL_ACCESS
0x128	File	C:\Windows\en-US\explorer.exe.mui	0xFFFFB80014658520	0x00120089	READ_CONTROL   SYNCHRONIZE   FILE_GENERIC_READ
0x1B0	File	\Device\DeviceApi	0xFFFFB800067C34C0	0x00120089	READ_CONTROL   SYNCHRONIZE   FILE_GENERIC_READ
0x1E4	File	\Device\KsecDD	0xFFFFB800154AC3F0	0x00100003	SYNCHRONIZE   WRITE_DATA   READ_DATA
0x1EC	Mutant	\Sessions\2\BaseNamedObjects\SMO:15524:304:WinStaging_02	0xFFFFB800F362BB70	0x001F0001	READ_CONTROL   DELETE   SYNCHRONIZE   WRITE_DAC   WRITE_OWNER   MUTAN
0x1F0	Semaphore	\Sessions\2\BaseNamedObjects\SMO:15524:304:WinStaging_02_p0	0xFFFFB800A0257F80	0x001F0003	READ_CONTROL   DELETE   SYNCHRONIZE   WRITE_DAC   WRITE_OWNER   SEMAPI
0x1F4	Semaphore	\Sessions\2\BaseNamedObjects\SMO:15524:304:WinStaging_02_p0h	0xFFFFB800A0256F30	0x001F0003	READ_CONTROL   DELETE   SYNCHRONIZE   WRITE_DAC   WRITE_OWNER   SEMAPI
0x2C0	Key	HKCU	0xFFFFD00A8B4E21C0	0x000F003F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   KEY_ALL_ACCESS
0x2CC	Key	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer	0xFFFFD00A8B4E22E0	0x000F003F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   KEY_ALL_ACCESS
0x2D0	Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Ids	0xFFFFD00A8B4E2520	0x00020019	READ_CONTROL   KEY_READ
0x320	Mutant	\Sessions\2\BaseNamedObjects\SessionInmersiveColorMutex	0xFFFFB800A8B243E0	0x001F0001	READ_CONTROL   DELETE   SYNCHRONIZE   WRITE_DAC   WRITE_OWNER   MUTAN
0x324	Section	\Sessions\2\BaseNamedObjects\SessionInmersiveColorPreference	0xFFFFD00AF13C6610	0x000F001F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   SECTION_ALL_ACCESS
0x328	File	C:\Windows\WinSxS\amd64_microsoft.windows.common-controls_6595...	0xFFFFB80010A45380	0x00100020	SYNCHRONIZE   EXECUTE
0x330	Key	HKCU\Software\Classes	0xFFFFD00A8B4E2640	0x000F003F	READ_CONTROL   DELETE   WRITE_DAC   WRITE_OWNER   KEY_ALL_ACCESS
0x334	Section	\BaseNamedObjects\_ComCatalogCache_	0xFFFFD00A8BAF40F0	0x00000004	MAP_READ

CPU Usage: 20.57% | Commit Charge: 81.31% | Processes: 314 | Physical Usage: 76.57%

Рисунок 1-10.Снимок экрана процесса «Проводник»

По умолчанию Process Explorer показывает только дескрипторы для объектов, которые имеют имена .

Чтобы просмотреть все маркеры в процессе, выберите «Show Unnamed Handles and Mappings from Process» из меню «View» в Process Explorer.

## 12) Доступ к существующим объектам

Столбец «Access» в представлении дескрипторов Process Explorer показывает маску доступа, которая использовалась для открытия или создания объекта. Эта маска доступа является ключом к тому, какие операции разрешено выполнять с определенным объектом.

Например, если клиентский код хочет завершить процесс, он должен вызвать сначала функцию `OpenProcess`, чтобы получить дескриптор требуемого процесса с маской доступа (минимум) `PROCESS_TERMINATE`, иначе не будет возможности завершить процесс с этим дескриптором.

Если вызов успешен, то вызов `TerminateProcess` обязательно будет успешным. Вот пример кода завершения процесса с указанным идентификатором процесса:

```
bool KillProcess(DWORD pid) {  
// open a powerful-enough handle to the process  
HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);  
if (!hProcess)  
return false;  
// now kill it with some arbitrary exit code  
BOOL success = TerminateProcess(hProcess, 1);  
// close the handle  
CloseHandle(hProcess);  
return success != FALSE;  
}
```



Столбец «Decoded Access» в Process Explorer, содержит текстовое описание маски доступа (для некоторых типов объектов), упрощая распознавание точного доступа, разрешенного для определенного дескриптора.

Двойной щелчок по записи дескриптора показывает некоторые свойства объекта. Рисунок 1-11 показывает свойства объекта.

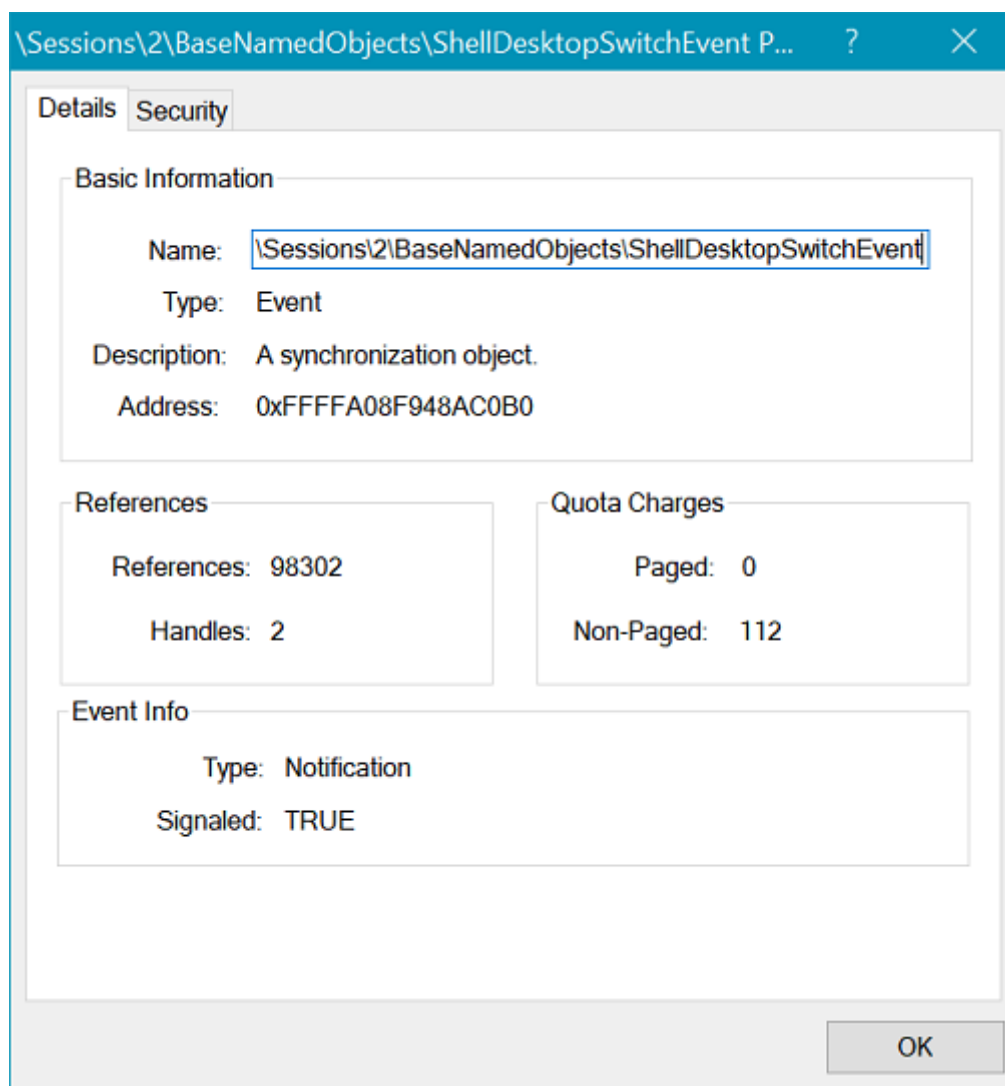


Рисунок 1-11.Свойства объекта

Свойства на рисунке 1-11 включают имя объекта (если есть), его тип, описание, его адрес в памяти ядра, число открытых дескрипторов и некоторая детальная информация об объекте, такая как состояние и тип отображаемого объекта события.

Обратите внимание, что показанные ссылки не указывают на фактическое количество ссылок на объект.

Правильный способ увидеть фактический счетчик ссылок для объекта, должен использовать команду!trueref отладчика ядра, как показано здесь:

```
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
```

Мы рассмотрим более подробно атрибуты объектов и отладчик ядра в следующих главах.

Теперь давайте начнем писать очень простой драйвер, чтобы показать и использовать многие инструменты, которые нам понадобятся позже в этой книге.

**ВАЖНО: В этой главе вкратце была описана архитектура ядра, без понимания всего изложенного в этой статье, нет смысла переходить к следующим главам.**

**Если вы что-то непоняли в этой статье, можно спросить у автора перевода, т. е. меня на форуме <https://ru-sfera.org/forums/windows-kernel-programming.161/> (Xe-xe).**

**Также можно читать оригинал книги.)))**