**Отладчик Windows:**

Отладчики режима ядра:

* Отладчик с графическим пользовательским интерфейсом - WinDBG
* Отладчик командной строки - KD

Режимы отладчика:

1. Live Debugging (прямая отладка)
   * Отладчик во время работы ОС соединяется с ней через порты via Serial/1394/USB2.0
   * Отладчик соединяется с ОС, что позволяет исследовать системную память
2. Отладка post mortem
   * Состояние ОС описано в crash dump файле
   * Отладчик загружает crash dump файл и начинает анализ причин сбоя

Компоненты отладчика:

* Пользовательский интерфейс (windbg.exe, kd.exe) - заменяемый; управляет пользовательским вводом/выводом данных.
* Ядро отладчика (dbgeng.dll) - обрабатывает действия отладчика и предоставляет API для расширений отладчика и пользовательского интерфейса.
* Обработчик символьной информации (dbghelp.dll) - находит табличную информацию о символе из .PDB файлов.

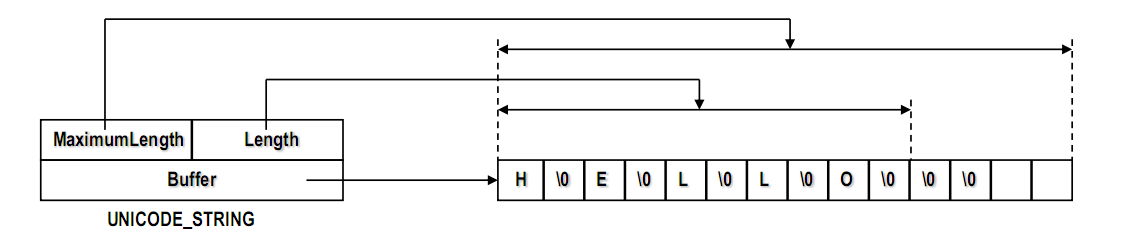
**Строки Unicode**

Ядро Windows хранит строки в формате Unicode. Строки, передаваемые функциям ядра, находятся почти всегда в формате Unicode. Для их представления используется структура UNICODE\_STRING

Структура UNICODE\_STRING:

Буфер, на который указывает UNICODE\_STRING.Buffer, обычно выделяется из пула подкачиваемой страничной памяти.

UNICODE\_STRING.Length содержит число байтов (не WCHARS). Завершающий символ UNICODE NULL в это число не включен.



Команды отладчика для просмотра строк:

* ‘du’ отображает символы Unicode
* ‘! du’ отображает физический адрес символов Unicode
* ‘.enable\_unicode’ позволяет/запрещает отладчику напечатать PUSHORTs как строки Unicode
* ‘dS’ отображает структуру UNICODE\_STRING

**Двусвязные списки**

Большинство структур данных ядра хранятся как части двусвязных списков. Структура LIST\_ENTRY используется для создания и обслуживания таких списков. Список состоит из поля ListHead (тип LIST\_ENTRY) и различного числа элементов списка (List Elements, включающих структуру типа LIST\_ENTRY).

Поля Flink & Blink в структуре LIST\_ENTRY указывают на вложенную структуру LIST\_ENTRY, а не на List Element.

Структуры LIST\_ENTRY \_никогда\_ не содержат нулевых указателей. Когда список пуст, поля Flink и Blink ListHead указывают непосредственно на ListHead.



**Уровни прерываний (IRQL // Interrupt ReQuest Level)**

Прерывания обрабатываются в порядке приоритетов. Прерывание с более высоким приоритетом может прервать обработчик прерывания с более низким приоритетом. Все запросы на прерывание более низкого приоритета маскируются до завершения обработки всех более приоритетных прерываний. Затем, если менее приоритетные прерывания происходили, они материализуются.

IRQL - приоритет, связанный с каждым прерыванием. Чем больше IRQL, тем выше приоритет прерывания.

Когда процессор обслуживает прерывание, IRQL процессора повышен.

Когда процессор находится на определенном IRQL, все последующие прерывания, равные либо ниже данного IRQL, “застревают” на этом же процессоре (в контроллере PIC).

На многопроцессорных системах каждый процессор может находиться на своем IRQL. Текущий IRQL процессора сохранен в Области Управления Процессором (KPCR – processor control block).

Отображение между IRQ и IRQLs определяет HAL. Обычно используется циклическое выделение IRQL для IRQ, отличающееся на многопроцессорных системах. Драйверы не могут изменить это отображение.

Код режима ядра может менять IRQL процессора.

Код пользовательского режима всегда выполняется на PASSIVE\_LEVEL.

Таблица приоритетов прерываний для процессоров x86:

|  |  |  |
| --- | --- | --- |
| **Уровень** | **IRQL** | **Назначение** |
| Уровни аппаратных прерываний | | |
| 31 | HIGH\_LEVEL | Machine check, NMI. Немаскируемое прерывание. Предназначено для остановки системы и маскирования всех прерываний. Вызывается из KeBugCheckEx. |
| 30 | POWER\_LEVEL | Power failure. Предназначено на тот случай, если происходит пропадание электропитания. Реально никогда не использовалось. |
| 29 | IPI\_LEVEL | Inter-processor interrupt. Используется для того, чтобы попросить другой процессор выполнить какие-то действия, например, обновить translation look-aside buffer (TLB) cache. |
| 28 | CLOCK\_LEVEL | Clock interrupt. Используется для обновления системного времени и учета выделяемого потокам времени. |
| 27 | PROFILE\_LEVEL | Profiling timer interrupt. Используется системным таймером для профилирования ядра, если включен такой режим. Собирается информация о выполняемых функциях путем анализа стека. |
| 3 – 26 | DEVICE\_LEVEL | DIRQLx – device interrupts. Уровни устройств. Они назначаются операционной системой. Алгоритм отличается на x86 (one CPU: 27 – IRQ number; multiple CPUs: round-robin в диапазоне IRQL устройств) и x64/IA64 (IRQ number / 16). |
| Уровни программных прерываний | | |
| 2 | DISPATCH\_LEVEL/  DPC\_LEVEL | Dispatch code and DPCs. Уровень программных прерываний. На этом уровне работает планировщик потоков (диспетчер). Повышение IRQL процессора до этого уровня прекращает переключение задач на данном процессоре. При этом планировщик все еще может работать на других процессорах. Обработчики аппаратных прерываний тоже могут работать, потому что их IRQL выше. Блокирование/ожидание на уровне DISPATCH\_LEVEL вызывает deadlock. К страничной памяти нельзя получить доступ на уровне IRQL >= DISPATCH\_LEVEL. Память, к которой выполняется обращение на уровнях прерываний DISPATCH\_LEVEL и выше, должна быть резидентной (non-paged). |
| 1 | APC\_LEVEL | APCs and page faults. На уровне APC\_LEVEL работает обработчик пробоя страницы. Применяются приоритеты потоков. Каждый поток может независимо устанавливать для себя уровень APC\_LEVEL, т.е. этот уровень программных прерываний локален для потока. |
| 0 | PASSIVE\_LEVEL/  LOW\_LEVEL | Thread switching. Фактически не является уровнем прерывания и существует для полноты. Уровень PASSIVE\_LEVEL означает обычную работу процессора при выполнении потока. На этом уровне применяются приоритеты потоков. Если поток выполняет системный вызов, который приводит к переходу из пользовательского режима в режим ядра, IRQL не изменяется. |

Таблица приоритетов прерываний для процессоров x64:

|  |  |
| --- | --- |
| **Уровень** | **IRQL** |
| 15 | HIGH\_LEVEL/ PROFILE\_LEVEL |
| 14 | POWER\_LEVEL/  IPI\_LEVEL |
| 13 | CLOCK\_LEVEL |
| 12 | SYNC\_LEVEL |
| 3 – 11 | DEVICE\_LEVEL |
| 2 | DISPATCH\_LEVEL/  DPC\_LEVEL |
| 1 | APC\_LEVEL |
| 0 | PASSIVE\_LEVEL/  LOW\_LEVEL |

Поток, работающий в ядре, может подымать и опускать уровень прерываний с помощью функций KeRaiseIrql() и KeLowerIrql(). Но чаще всего это происходит неявно при работе с объектами синхронизации. При блокировке объекта происходит поднятие уровня до соответствующего объекту значения. При освобождении объекта выполняется снижение уровня. Функции KeRaiseIrql() и KeLowerIrql() реализованы в HAL. В реализации используется оптимизация под названием lazy IRQL.

При написании обработчиков прерываний следует учитывать, что на уровнях DPC/Dispatch и выше нельзя выполнять ожидание объектов, если это требует переключения процессора на другой поток. Дело в том, что переключение на другой поток выполняется планировщиком на уровне DPC/Dispatch, который в данном случае замаскирован, и поэтому возникает блокировка. Отсюда следует, что в обработчиках прерываний на уровнях DPC/Dispatch и выше можно работать лишь с физической (non-paged) памятью. Объясняется это тем, что попытка доступа к странице, которой нет в памяти, вызывает прерывание, в ответ на которое менеджер памяти вынужден инициировать подкачку страницы с диска и дождаться завершения операции. Ожидание означает переключение на другой поток через вызов прерывания уровня DPC/Dispatch, но оно замаскировано. Нарушение правила приводит к тому, что система обваливается с кодом IRQL\_NOT\_LESS\_OR\_EQUAL. В библиотеке WDK существует программа Driver Verifier, которая позволяет выявить ошибки такого рода. Библиотека WDK определяет допустимый уровень IRQLs для каждой API-функции ядра. Драйверы устройств должны придерживаться этих рекомендаций.

|  |
| --- |
| IRQL API: |
| KeRaiseIrql()  KeLowerIrql()  KeGetCurrentIrql()  KeRaiseIrqlToDpcLevel() |

**Любой код в режиме ядра работает в одном из следующих контекстов:**

* Процедура обработки прерывания (ISR)
  + Выполняется в DIRQLx в произвольном контексте потока
  + Связана с аппаратными прерываниями
* Отложенный вызов процедуры (DPC)
  + Выполняется на DISPATCH\_LEVEL в произвольном контексте потока
  + Выполняет задержанную обработку прерываний на низком IRQL
* Асинхронный вызов процедуры (APC)
  + Выполняется на APC\_LEVEL или PASSIVE\_LEVEL в определенном контексте потока
* Контекст Потока Пользовательского Режима
  + Выполняется на PASSIVE\_LEVEL в пользовательском режиме в контексте потока
  + Потоки, переходившие к режиму ядра для обработки системного вызова
* Контекст Потока Режима Ядра
  + NTOSKRNL & Driver Worker Thread Context
    - Выполняется на уровне PASSIVE\_LEVEL в контексте системного потока
    - Пользовательские потоки, создаваемые NTOSKRNL или драйверами устройств для личных нужд.
  + Контекст рабочего системного потока
    - Выполняется на уровне PASSIVE\_LEVEL в контексте системного потока
    - Исполняющие подпрограммы, вызванные системными исполняющими потоками ExpWorkerThread ()

**Процедура обработки прерывания (ISR)**

Аппаратные прерывания вызываются аппаратурой. Программист, разрабатывающий драйвер, может определить свой обработчик аппаратного прерывания. Но для этого нужно представлять цепочку событий в системе и структуры данных, которые в этой цепочке участвуют.

Периферийное устройство –> Контроллер прерываний –> IDT –> Объект прерывания (ISR-адрес, IRQL, Spinlock, Dispatch code – копируется из шаблона KiInterruptTemplate) –> KiInterruptDispatch или KiChainedDispatch (raise IRQL, grab Spinlock, call Driver ISR, drop Spinlock, lower IRQL) –> Driver ISR (read from device, acknowledge-interrupt, request DPC).

При написании драйвера бывает нужно подсоединить-отсоединить свой обработчик прерывания. Это делается с помощью функций IoConnectInterrupt() и IoDisconnectInterrupt().

**Таблица дескрипторов прерываний**

**(Interrupt Dispatch Table)**



* Прерывания сгенерированы:
  + Устройствами для уведомления о событии
  + Компонентами программного обеспечения для вызовов ОС
* Исключения происходят, когда выполнение

команд генерирует событие, которое не может быть стандартно обработано ЦП

* Элементы Таблицы содержат адреса обработчиков прерываний и обработчиков исключений
  + Ядро инициализирует IDT
  + ЦП ищет IDT и диспетчеризирует обработчики
* Записи IDT (индексированные векторами) классифицированы как:
  + Исключения (0x00 – 0x1f)
    - Синхронны с потоком команд ЦП
    - Примеры: Divide By Zero, Page Fault, Access Violation
  + Программные прерывания (0x2a – 0x2e)
    - Прерывания Операционной системы (вызывающие службы ОС)
    - Примеры: int 2e(KiSystemService), int 2A(KiGetTickCount)
  + Аппаратные прерывания (0x31 – 0xfd)
    - Асинхронны с потоком команд ЦП
    - Примеры: NIC, RTC, Клавиатура, Звуковые карты, Serial Port, IPI, Profile, APIC Error, ACPI и т.д.

**Отложенный вызов процедуры (Deferred Procedure Call)**

* ISRs(процедуры обработки прерывания), работающие на определенном IRQL (DIRQLx), могут предотвратить выполнение других ISR на более нижних IRQLs
  + - Это увеличивает время ожидания для системных прерываний
* Чтобы уменьшить Задержку прерывания, ISRs:
  + - Выполняют абсолютный минимум работ на DIRQLx
    - Запрашивают DPC выполнить объем работы на DISPATCH\_LEVEL ()
    - Позволяют системе быстро обработать другие прерывания
  + DPCs используются, чтобы задержать обработку прерывания на более нижних IRQLs ()
  + Структура KDPC представляет DPC
    - Содержит функцию повторного вызова (подпрограмма DPC)
    - Структура KDPC поставлена в очередь в специальный список DPC CPU в KPCR.PrcbData.DpcListHead
  + Подпрограммы DPC выполняются:
    - В произвольном контексте потока в IRQL == DISPATCH\_LEVEL
    - Используя специальный стек ЦП, указатель которого в KPCR.PrcbData.DpcStack



|  |
| --- |
| DPC API (Kernel) |
| KeInitializeDpc()  KeInsertQueueDpc()  KeRemoveQueueDpc()  KeSetTargetProcessorDpc()  KeSetImportanceDpc()  KeFlushQueuedDpcs() |

**Асинхронный вызов процедуры (Asynchronous Procedure Call)**



|  |  |
| --- | --- |
| APC API | |
| Kernel | User |
| KeInitializeApc()  KeInsertQueueApc()  KeRemoveQueueApc()  KeFlushQueueApc()  KeAreApcsDisabled()  KeEnterCriticalRegion()  KeLeaveCriticalRegion() | QueueUserApc() |

* Используется для выполнения функции в контексте определенного потока
* APC, выполненные на PASSIVE\_LEVEL или APC\_LEVEL
* KTHREAD.ApcState. ApcListHead [] содержит список APCs, ожидающих потока
  + Содержит начала списка (list heads) UserMode & KernelMode
* Структура KAPC представляет APC
  + KAPC вставлен в Очередь потока APC
  + Организация очередей APC не вызывает поток к ожиданию аварийного завершения
    - Кроме случаев, когда target поток ждет в

alertable состоянии ожидания в обращении к WaitForSingleObjectEx (), SleepEx () и т.д.

* + - Подпрограмма APC имеет приоритет по non-APC коду
* Подпрограммы KAPC
  + RundownRoutine – Выполняется, когда APC отброшен во время завершения потока
  + KernelRoutine – Выполняется на APC\_LEVEL
  + NormalRoutine – Выполняется на PASSIVE\_LEVEL

**Типы APC(режим ядра)**

* Специальный режим ядра
  + KAPC.KernelRoutine выполняется на APC\_LEVEL
  + KAPC.NormalRoutine == NULL
  + Delivered перед APCs нормального режима ядра
    - Вставлен в очередь APC потоков после других специальных APCs
  + Deliverable, если
    - IRQL == PASSIVE\_LEVEL
    - Поток не находится в защищенной области (KTHREAD.SpecialApcDisable! = 0)
  + Подпрограмма APC не может получить мьютексы, доступные в IRQL == 0
  + Используется для завершения процесса
* Нормальный Режим Ядра
  + KAPC.NormalRoutine на PASSIVE\_LEVEL
  + Deliverable, если
    - IRQL == PASSIVE\_LEVEL
    - Поток не находится в критической секции (KTHREAD.KernelApcDisable! = 0)
    - Поток не выполняет APC ядра
  + Подпрограмме APC разрешено делать все системные вызовы
  + Используется ОС для завершения IRP

**Типы APC (Пользовательский режим)**

* Пользовательский режим
  + - KAPC.NormalRoutine выполняется на PASSIVE\_LEVEL
    - Deliverable, только если поток находится в alertable состоянии ожидания
    - Поток может ввести alertable состояние ожидания, вызвав
      * WaitForSingleObectEx с Alertable = TRUE
      * WaitForMultipleObjectsEx с Alertable = TRUE
    - Queued через подтверждение посредством API Win32 (QueueUserApc())
    - Используется ОС, чтобы выполнить подпрограммы завершения ввода-вывода пользовательского режима
  + KAPC режима ядра/пользовательского режима может содержать опцию KAPC.KernelRoutine
    - Выполняется в IRQL == APC\_LEVEL
    - Выполняется перед NormalRoutine
    - Для APCs пользовательского режима KAPC.KernelRoutine и KAPC.NormalRoutine выполняются только, когда поток находится в пользовательском режиме alertable ожидания

**Единицы работы**

* В ОС предусмотрен механизм to execute a custom driver provided

function на PASSIVE\_LEVEL в контексте системного потока

* + - WORK\_QUEUE\_ITEM использовался в NT4 и предшествующих
    - IO\_WORKITEMs были представлены в Win2K
      * Ссылка устанавливается на объект устройства так, чтобы она не была удалена, пока выполняется рабочая единица подпрограммы



**Расположение Виртуального Адресного Пространства Режима Ядра**

2 GB / 3 GB

0x 8000 0000

or

0x c000 0000

|  |
| --- |
| NTOSKRNL Image |
| HAL Image |
| PFN Database |
| Non Paged Pool |
| Session Pool |
| Session View |
| Session Image |
| Page Table Entries |
| System Cache |
| Paged Pool |
| Driver Image |
| Free |
| Kernel Stack |
| Kernel Stack |
| Driver Image |
| Kernel Stack |
| Kernel Stack |
| Free |
| Non Paged Pool Expansion |

4 GB

0x ffff ffff

**Пулы памяти**

* Выделение динамической памяти в режиме ядра осуществляется из пулов памяти, то есть из хипов в режиме ядра
  + - Страничный пул – страничная память
    - Нестраничный пул – page locked физически фрагментированная память

(такой пул используется обработчиками прерываний)

* + - Session Pool – особый страничный пул, в котором располагаются данные сессии
    - Специальный пул – страничный/нестраничный с возможностью обнаружения нарушения целостности данных (используется для отладки)
    - Страничные/нестраничные пулы растут до установленного максимума системы
    - Максимальный размер пулов задается при системном запуске в реестре на основании количества физической памяти в системе
* У системы есть multiple страничные пулы во избежание сериализации вызовов пула
* При выделения памяти для пула можно передать 4-х символьный тег (которым будет помечен выделенный блок памяти на границе страницы. Обычно тег соответствует названию драйвера, выделившего память.)
  + - Это полезно для отладки повреждений и утечек данных пула

|  |
| --- |
| Pool API (Kernel) |
| ExAllocatePoolWithTag()  ExAllocatePoolWithQuota()  ExAllocatePoolWithQuotaTag()  ExFreePoolWithTag()  ExFreePoolWithQuotaTag() |

|  |  |
| --- | --- |
| Registry entries that control pool sizes (used override system determined default values) | |
| HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management | |
| NonPagedPoolQuota | Maximum non-paged pool thet can can be allocated to a process |
| NonPagedPoolSize | Size of non-paged pool (default == 0) |
| PagedPoolQuota | Maximum paged pool that can be allocated to a single process |
| PagedPoolSize | Size of non-paged pool (default == 0 ; 0xffffffff == max possible) |
| SesionPoolSize | Size of Session Paged Pool in MB. Used by Display Driver |

**Структура данных пула**

* Система содержит структуру POOL\_DESCRIPTOR для каждого пула
* Переменная PoolVector содержит массив указателей, которые указывают на дескрипторы страничных/нестраничных пулов
  + - Точка входа страничного пула указывает на массив дескрипторов пула
    - Точка входа нестраничного пула указывает на единственный дескриптор пула
* Блокам пула предшествует структура POOL\_HEADER которая содержит отслеживаемую информацию о выделенных и свободных блоках
  + - POOL\_HEADER содержит:
      * Тег пула для блока пула
      * Указатель EPROCESS, который является квотой, взимаемой за блок пула
      * Индекс пула, которому блок принадлежит (только для страничных пулов)
      * Размер блока пула и размер предыдущего блока



**Расположение объектов**

* Данные объектов хранятся в нестраничном пуле
* Диспетчер объектов использует известные теги для объектов
  + - Может использоваться для идентификации объектов в пуле
    - Например, у объектов файла есть тег 'Файл'
* Дескрипторы защиты размещаются в страничном пуле
  + - Указатель на дескриптор защиты хранится в заголовке объекта
* Таблица указателей размещается в страничном пуле. Каждый элемент таблицы:
  + - описывается типом HANDLE\_TABLE\_ENTRY
    - содержит указатель на объект и биты Granted Access (с разрешенным доступом)



**Ассоциативные(резервные) списки (Lookaside Lists)**

Диспетчер куч (heap manager) управляет кучами, как системными, так и пользовательскими, разбивая пространство кучи на блоки и организуя списки блоков одинакового размера. Если приходит запрос на выделение блока памяти из кучи, то диспетчер куч пытается подобрать свободный блок подходящего размера. На это, естественно, требуется какое-то время. Если же заранее известно, что потребуются блоки памяти фиксированного размера, но количество этих блоков и частота их использования не известны, то следует использовать, по соображениям лучшей производительности, так называемые ассоциативные списки (look-aside lists), которые существуют только в ядре. Главное отличие ассоциативных списков от пулов в том, что из ассоциативных списков можно выделять блоки памяти только фиксированного и заранее определенного размера, а из пулов любого, причем память из ассоциативных списков выделяется быстрее, так как нет необходимости подбирать подходящую область свободной памяти. Стоит отметить, что при многократном выделении памяти из пулов возникает проблема фрагментированности.

Перевод английского термина look-aside list на русский язык крайне абстрактен и плохо отражает суть. Look-aside дословно можно перевести как: "смотреть по сторонам". Смысл с том, что look-aside list представляет собой набор или список заранее выделенных системой блоков памяти. В каждый момент времени какие-то блоки могут быть свободны, а какие-то заняты. При запросе на выделение очередного блока задача системы пройти по списку ("посмотреть по сторонам ") и найти близлежащий свободный блок. В русском же переводе, look-aside превращается в "ассоциативный", и становится совершенно непонятно, что с чем здесь ассоциируется. Тем не менее, перевод этот устоявшийся - хочешь, не хочешь, придется применять. Т.о. ассоциативный список - это фактически особая системная куча, работающая по определенным правилам.

При инициализации ассоциативного списка он создается пустым, но мы нигде не указываем, сколько всего блоков памяти нам может потребоваться. Откуда же система знает, сколько памяти она должна заблаговременно выделить? Ведь если этого не сделать заранее, то ассоциативный список не будет работать быстрее, чем системный пул. Дело в том, что изначально система выделяет всего несколько блоков, количество которых определяется самой системой. Если теперь мы начнем выделять память из ассоциативного списка, то будут возвращаться указатели именно на эти заранее выделенные блоки. Один раз в секунду система проводит настройку всех ассоциативных списков в системе, вызывая функцию ExAdjustLookasideDepth. Если при настройке ассоциативного списка система увидит, что резерв свободных блоков уменьшился, то выделит новые. Количество вновь выделяемых блоков зависит от нагрузки на ассоциативный список, т.е. от частоты с которой мы выделяем из него память. Система пытается настроить ассоциативный список так, чтобы он работал наиболее эффективно. Если в интервале времени между двумя настройками мы исчерпали лимит заранее выделенных блоков, то система начинает использовать системный пул до следующей настройки, когда она увидит, что резерв равен нулю и выделит необходимый лимит. Здесь важно понять, что если скорость выделения блоков уж слишком велика, то можно и не получить выигрыша в производительности по сравнению с системным пулом. В этом случае придется организовывать собственную кучу. Оценить насколько эффективно работает ассоциативный список можно с помощью команды !lookaside отладчика MS Kernel Debugger.

Все lookaside списки определенного типа поддерживаются в служебном связном списке

* + - ExPagedLookasideListHead – список страничных ассоциативных списков
    - ExNPagedLookasideListHead – список нестраничных ассоциативных списков
    - Allocation из ассоциативных списков:
      * разрешено, при использовании распределяющих функций пула, если список пуст
      * разрешено, при наличии блоков, которые уже находятся в ассоциативных списках, если список не пуст
    - Освобождение блока памяти для ассоциативного списка не делает блок свободным для пула, однако ставит его в очередь ассоциативного списка
* Все ассоциативные списки определенного типа поддерживаются в связном списке

|  |
| --- |
| Lookaside List API (Kernel) |
| ExAllocateFromPagedLookasideList()  ExInitializePagedLookasideList()  ExFreeToPagedLookasideList()  ExDeletePagedLookasideList()  ExAllocateFromNPagedLookasideList()  ExInitializeNPagedLookasideList()  ExFreeToNPagedLookasideList()  ExDeleteNPagedLookasideList() |

**Фиксация данных в физической памяти**

* Компоненты режима ядра(драйверы устройств) могут получать нестраничную память следующими способами:
  + - выделить непрерывный блок, зафиксированный в нестраничной памяти
      * это очень дорогой системный ресурс и должен использоваться экономно
    - выделить несколько небольших блоков нестраничной памяти
      * например, из нестраничного пула
    - выделить страничную память и заблокировать соответствующие ей страницы физической памяти
      * такая блокировка вызывает увеличение количества ссылок в PFN(page frame number) Entry страницы
    - отобразить память устройства в виртуальное адресное пространство ОС
      * отображенная память устройства всегда нестраничная
* Таблица описания памяти (Memory Descriptor List (MDL)) - структура данных системы, которая описывает отображение виртуальной памяти на физическую
* MDLs используются:
  + - чтобы отобразить нестраничную память
    - драйверами, для прямого доступ к памяти DMA (Direct Memory Access)

|  |
| --- |
| Non-Paged Memory API (Kernel) |
| MmAllocateContiguousMemory()  MmGetPhysicalAddress()  MmLockPageableCodeSection()  MmLockPageableDataSection()  MmLockPageableSectionByHandle()  MmUnlockPageableImageSection()  MmPageEntireDriver()  MmResetDriverPaging()  MmMapIoSpace() |

**Таблица описания памяти (MDL)**

MDL представляется одноименной структурой и содержит описание физических страниц региона памяти.

MDL STRUCT

Next PVOID ?

\_Size SWORD ?

MdlFlags SWORD ?

Process PVOID ?

MappedSystemVa PVOID ?

StartVa PVOID ?

ByteCount DWORD ?

ByteOffset DWORD ?

MDL ENDS

PMDL typedef PTR MDL

Точнее говоря, структура MDL это заголовок. Сразу за заголовком идет массив двойных слов Pages, каждое из которых представляет собой номер физической страницы (page frame number, PFN). Хотя регион памяти описываемый MDL является непрерывным в виртуальном пространстве адресов, занимаемые им физические страницы могут располагаться в физической памяти в произвольном порядке. Именно поэтому и нужен массив Pages, содержащий список всех физических страниц занимаемых регионом памяти. Также он требуется для организации прямого доступа к памяти (Direct Memory Access, DMA).

Функция IoAllocateMdl выделяет память для MDL и инициализирует его заголовок.

PMDL IoAllocateMdl(

\_In\_opt\_     PVOID VirtualAddress,

\_In\_         ULONG Length,

\_In\_         BOOLEAN SecondaryBuffer,

\_In\_         BOOLEAN ChargeQuota,

\_Inout\_opt\_  PIRP Irp

);

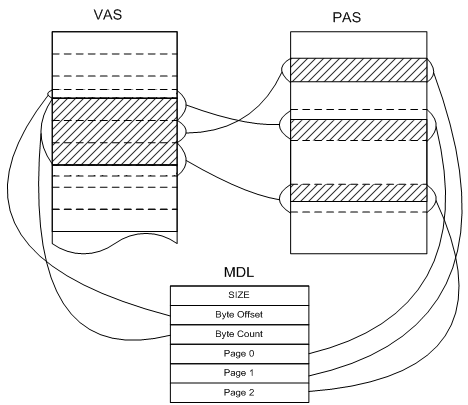
Первые два параметра функции IoAllocateMdl определяют виртуальный адрес и размер блока памяти, для которого надо сформировать MDL. Если MDL не ассоциируется с IRP, то третий параметр равен FALSE. Четвертый параметр определяет, нужно ли уменьшить квоту процесса, и применим только для драйверов находящихся на самом верхнем уровне в цепочке драйверов или для одноуровневых драйверов. Каждый процесс получает от системы квоты на ресурсы. Когда процесс выделяет себе ресурс, квота уменьшается. Если квота кончается, то и соответствующий ресурс больше не выделяется. Последний параметр определяет необязательный указатель на IRP, с которым ассоциируется MDL. Например, при прямом вводе-выводе диспетчер ввода-вывода создает MDL для пользовательского буфера и передает его адрес в IRP.MdlAddress

* Буфер описания MDL
  + - Не должен запускаться на границе страницы
      * MDL.ByteOffset - смещение запускающегося VA с начала первой страницы
    - Длина не должна зависеть от размера страницы
      * MDL.ByteCount - фактическое число байтов, описанных MDL
* MDL может описать страницы ядра/пользователя
* Буфер, описанный MDL, может быть отображен

от пользователя к VAS ядра или наоборот

|  |
| --- |
| IO MDL API (Kernel) |
| IoAllocateMdl()  IoBuildPartialMdl()  IoFreeMdl() |

|  |
| --- |
| MM MDL API (Kernel) |
| MmBuildMdlForNonPagedPool()  MmProbeAndLockPages()  MmUnlockPages()  MmPrepareForReuse()  MmMapLockedPages()  MmMapLockedPagesSpecifyCache()  MmUnmapLockedPages() |



**Точки входа драйвера устройства**



**Объект драйвера (DRIVER\_OBJECT)**

* Представляет отображение драйвера в памяти
  + - Только один экземпляр драйвера может быть загружен в системную память
    - Драйверы должны быть в состоянии поддерживать многократные экземпляры аппаратных средств
* Выделенный, когда драйвер загружен в память и освобожден, когда драйвер разгружен из памяти
* Содержит следующее:
  + - Список устройств создан драйвером
    - Указатели на точки входа драйверов
    - Указатель на имя службы, соответствующей этому драйверу
      * Это имя может отличаться от двоичного имени файла драйвера
    - Начальный адрес и размер двухуровневого изображения драйвера в памяти
* Указатель на объект драйвера передают к DriverEntry () и AddDevice () подпрограммы



**Объект устройства (DEVICE\_OBJECT)**

typedef struct \_DEVICE\_OBJECT {

CSHORT                      Type;

USHORT                      Size;

LONG                        ReferenceCount;

struct \_DRIVER\_OBJECT  \*DriverObject;

struct \_DEVICE\_OBJECT  \*NextDevice;

struct \_DEVICE\_OBJECT  \*AttachedDevice;

struct \_IRP  \*CurrentIrp;

PIO\_TIMER                   Timer;

ULONG                       Flags;

ULONG                       Characteristics;

\_\_volatile PVPB             Vpb;

PVOID                       DeviceExtension;

DEVICE\_TYPE                 DeviceType;

CCHAR                       StackSize;

union {

LIST\_ENTRY         ListEntry;

WAIT\_CONTEXT\_BLOCK Wcb;

} Queue;

ULONG                       AlignmentRequirement;

KDEVICE\_QUEUE               DeviceQueue;

KDPC                        Dpc;

ULONG                       ActiveThreadCount;

PSECURITY\_DESCRIPTOR        SecurityDescriptor;

KEVENT                      DeviceLock;

USHORT                      SectorSize;

USHORT                      Spare1;

struct \_DEVOBJ\_EXTENSION \*  DeviceObjectExtension;

PVOID                       Reserved;

} DEVICE\_OBJECT, \*PDEVICE\_OBJECT;

Объект устройства представляет физические аппаратные средства или логическое устройство.

* Состоит из 2 частей:
  + - Заголовка фиксированной длины: DEVICE\_OBJECT (определенная система)
    - Расширения переменной длины: Device extension (определенный драйвер)
* Часть цепочки DOs создана драйвером
* Может иметь/не иметь имя
  + - Имя может быть обеспечено создателем
    - Имя может быть автоматически сгенерировано ОС
      * FILE\_AUTOGENERATED\_DEVICE\_NAME
* Создается драйверами устройств вызовом IoCreateDevice ()
* Удаляется драйверами устройств вызовом IoDeleteDevice ()

Поле DriverObject (PDRIVER\_OBJECT) указывает на объект драйвера, связанный с этим устройством. Обычно это тот драйвер, который и создал данный объект устройства вызовом функции IoCreateDevice.

Поле NextDevice (PDEVICE\_OBJECT) указывает на следующий объект устройства, принадлежащий тому же драйверу, что и данный объект. Это поле объединяет объекты устройств драйвера в связный список, начинающийся с объекта, указанного в поле DeviceObject объекта драйвера.

Поле AttachedDevice указывает на объект устройства, который является многоуровневым



**Объект файла (FILE\_OBJECT)**

* Представляет открытый экземпляр объекта устройства
  + - FO ссылается на DO, много FOs могут ссылаться на один DO
    - DO не ссылается ни на один FO
* Создание FO: CreateFile ()/ZwCreateFile ()
* Удаление FO: CloseHandle ()/ZwClose ()
* FO-> FsContext и FO-> FsContext2 обычно хранить особую информацию объекта файла, требуемую драйверами
* поле FO-> FileName указывает на часть имени файла, интерпретируемую драйвером устройства или драйвером файловой системы (FSD)
* FILE\_OBJECTs для FSDs указывают на блок параметров тома(VPB)
  + - VPB представляет отношение между файловой системой и смонтированным томам
* Поле FO-> Event используется, чтобы блокировать обращения потоков с синхронными запросами, которые обрабатываются асинхронно драйверами



**Объектные Отношения**

* DRIVER\_OBJECT – представляет изображение драйвера устройства
  + - Содержит главу списка объектов устройства, создаваемых драйвером
* DEVICE\_OBJECT – представляет логическое или физическое устройство
  + - Содержит указатель на объект драйвера, который создал его
    - Часть списка объектов устройства в драйвере
* FILE\_OBJECT – представляет открытый экземпляр устройства
  + - Содержит указатель на объект устройства, открытый экземпляр которого он представляет



**Пакет запросов на ввод-вывод (IRP)**

I\O Request Packet – полуоткрытая структура

typedef struct \_IRP {

.

.

PMDL MdlAddress;

ULONG Flags;

union {

struct \_IRP \*MasterIrp;

.

.

PVOID SystemBuffer;

} AssociatedIrp;

.

.

IO\_STATUS\_BLOCK IoStatus;

KPROCESSOR\_MODE RequestorMode;

BOOLEAN PendingReturned;

.

.

BOOLEAN Cancel;

KIRQL CancelIrql;

.

.

PDRIVER\_CANCEL CancelRoutine;

PVOID UserBuffer;

union {

struct {

.

.

union {

KDEVICE\_QUEUE\_ENTRY DeviceQueueEntry;

struct {

PVOID DriverContext[4];

};

};

.

.

PETHREAD Thread;

.

.

LIST\_ENTRY ListEntry;

.

.

} Overlay;

.

.

} Tail;

} IRP, \*PIRP;

IRP описывает запрос ввода-вывода. Структура содержит заголовок фиксированной длины (тело самой структуры) и одного или нескольких блоков стека IO\_STACK\_LOCATION. Тело IRP хранит общую информацию о запросе ввода-вывода: указатели на буферы, данные о состоянии и др. Блоки стека содержат информацию специфичную для конкретного этапа обработки IRP. Передавая IRP на обработку драйверу, диспетчер в/в (или драйвер самостоятельно создающий IRP) заполняет верхний блок стека. Если драйвер, получивший IRP, решает отправить его на дальнейшую обработку нижестоящему драйверу, он заполняет следующий блок стека и передает IRP ниже и т.д. Т.о. блоки стека - по одному на каждый вызываемый драйвер - хранят информацию, необходимую каждому драйверу для обработки своей части запроса.

* Выделен диспетчером ввода-вывода в результате запроса приложения пользовательского режима или запроса драйвера
  + - Может быть выделен драйверами, которые будут диспетчеризированы другим драйверам DeviceExtension
    - Выделен от нестраничных ассоциированных списков
* IoCompleteRequest () освобождает IRP после завершения использования драйвером
* IoCallDriver () передает IRP драйверам
  + - Драйвер получает IRP, чтобы диспетчеризировать точки входа

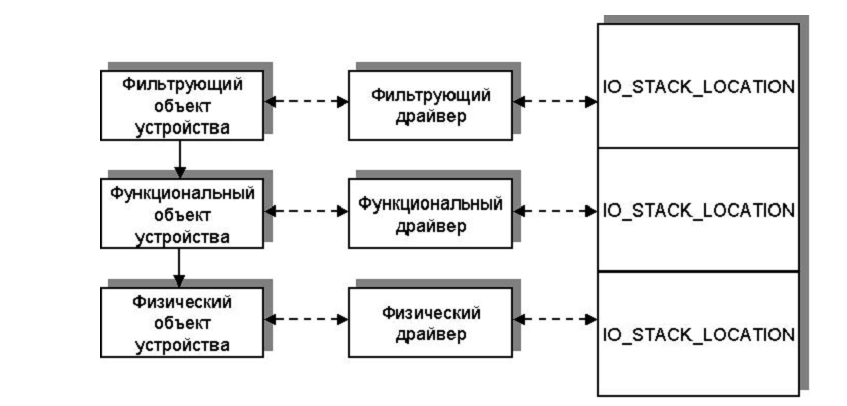


**Расположение Стека ввода-вывода (IO\_STACK\_LOCATION)**

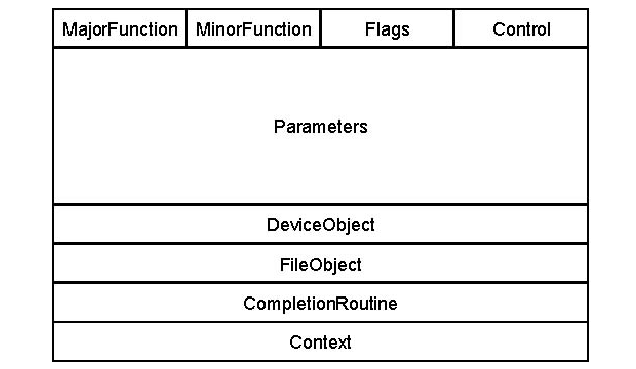
При создании IRP диспетчер ввода-вывода создает также массив структур

IO\_STACK\_LOCATION. В стеке содержится столько таких структур, сколько

драйверов будет обрабатывать данный IRP.



Каждый экземпляр IO\_STACK\_LOCATION содержит параметры ввода-вывода для определенного драйвера в стеке многоуровневых драйверов. Инициализируется предыдущим драйвером в стеке. Сначала IOSL инициализирован Диспетчером ввода-вывода



Поле MajorFunction (UCHAR) содержит основной код (значение) функции, связанной с IRP. Этот код, например, IRP\_MJ\_READ соответствует одному из элементов таблицы функций MajorFunction объекта драйвера.

Поле MinorFunction (UCHAR) содержит дополнительный код функции, который уточняет смысл пакетов IRP, если это необходимо. Например, запросы IRP\_MJ\_PNP подразделяются на подтипы при помощи дополнительных кодов функций, например, IRP\_MN\_START\_DEVICE.

Поле Flags содержит флаги обработки, определенные для выполняемой функции ввода/вывода. Это поле важно для драйверов файловых систем.

Поле Control является набором флагов, которые устанавливаются и читаются диспетчером ввода-вывода, указывая, как надо обработать данный пакет IRP. Например, в этом поле с помощью обращения драйвера к функции loMarklrpPending может быть установлен бит SL\_PENDING, указывающий диспетчеру ввода-вывода, что завершение обработки пакета IRP отложено.

Поле Parameters является объединением подструктур для каждого типа запросов, сопровождающихся заданием параметров. Например, подструктура Create нужна для запросов IRP\_MJ\_CREATE, подструктура Read нужна для запросов IRP\_MJ\_READ и т. д.

Поле DeviceObject (PDEVICE\_OBJECT) указывает на объект устройства, соответствующего данной позиции стека. Для заполнения поля вызывается функция IoCallDriver.

Поле FileObject (PFILE\_OBJECT) содержит адрес файла режима ядра, которому направляется IRP. Этот указатель часто используется для отмены всех пакетов IRP при закрытии соответствующего файла.

Поле CompletionRoutine (PIO\_COMPLETION\_ROUTINE) содержит адрес функции завершения ввода-вывода. Эта функция устанавливается вышестоящим (в стеке) драйвером.

Поле Context (PVOID) содержит то, что передается в качестве аргумента функции завершения.

