1. Модель программного интерфейса операционной системы Windows. Нотация программного интерфейса. Понятие объекта ядра и описателя объекта ядра операционной системы Windows. Модель архитектуры ОС Windows.

Интерфейс прикладного программирования Windows API (application programming interface) является интерфейсом системного программирования в пользовательском режиме для семейства операционных систем Windows.

API - набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

- **Процедурный API.** Единая точка доступа к службе – за вызовом процедуры стоит программное прерывание.
- **Объектный подход**. Отсутствие указателей на внутренние структуры данных ОС. Применение описателей (дескрипторов) вместо указателей.
- «**Венгерская» нотация в идентификаторах.**

Суть венгерской нотации сводится к тому, что имена идентификаторов предваряются заранее оговорёнными префиксами, состоящими из одного или нескольких символов. При этом, как правило, ни само наличие префиксов, ни их написание не являются требованием языков программирования, и у каждого программиста (или коллектива программистов) они могут быть своими.

Понятие объекта ядра и описателя объекта ядра операционной системы Windows.

Система позволяет создавать и оперировать с несколькими типами объектов ядра, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидаемыми таймерами (waitable timer objects). Эти объекты создаются Windows-функциями. Каждый **объект ядра** — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия. Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений. Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют с объектами ядра? Ответ в том, что в Windows предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда Вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. **Описатель** следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком Вашего процесса. Этот описатель Вы передаете Windows-функциям, сообщая системе, какой объект ядра Вас интересует.

В Windows входят следующие компоненты, работающие в режиме ядра: 1. **Исполняющая система** 2. **Ядро Windows** 3. **драйверам устройств** 4. **Уровень аппаратных абстракций** 5. **Система организации многооконного интерфейса и графики**,

Модель архитектуры ОС Windows. У вспомогательных системных процессов, у процессов служб, у пользовательских приложений и у подсистем среды окружения, — у всех есть свое собственное закрытое адресное пространство.

Четырем основным процессам пользовательского режима можно дать следующие описания:

1. **Фиксированные** (или реализованные на аппаратном уровне) **вспомогательные системные процессы**, такие как процесс входа в систему и администратор сеансов — Session Manager, которые не входят в службы Windows (они не запускаются диспетчером управления службами).2. **Служебные**
3. **Пользовательские приложения**4. **Серверные процессы подсистемы окружения**, которые реализуют часть поддержки среды операционной системы или специализированную часть

2. Понятие пользовательского режима и режима ядра операционной системы Windows. Модель виртуальной памяти процесса в пользовательском режиме и в режиме ядра операционной системы Windows. Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows. Основные модули ОС Windows.

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в Windows используются два процессорных режима доступа (даже если процессор, на котором работает Windows, поддерживает более двух режимов): пользовательский режим и режим ядра. Код пользовательского приложения запускается в **пользовательском режиме**, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра. **Режим ядра** — такой режим работы процессора, в котором предоставляется доступ ко всей системной памяти и ко всем инструкциям центрального процессора. Предоставляя программному обеспечению операционной системы более высокий уровень привилегий, нежели прикладному программному обеспечению, процессор гарантирует, что приложения с неправильным поведением не смогут в целом нарушить стабильность работы системы. Хотя у каждого Windows-процесса есть свое собственное закрытое адресное пространство, код операционной системы и код драйвера устройства, используют одно и то же общее виртуальное адресное пространство. Каждая страница **в виртуальной памяти** имеет пометку, показывающую, в каком режиме доступа должен быть процессор для чтения и (или) записи страницы. Доступ к страницам в системном пространстве может быть осуществлен только из режима ядра, тогда как доступ ко всем страницам в пользовательском адресном пространстве может быть осуществлен из пользовательского режима

Виртуальная память

В Windows реализована система виртуальной памяти, которая образует плоское (линейное) адресное пространство. Она создает каждому процессу иллюзию того, что у него есть достаточно большое и закрытое от других процессов адресное пространство. Виртуальная память дает логическое представление, которое не обязательно соответствует структуре физической памяти. В период выполнения диспетчер памяти, используя аппаратную поддержку, транслирует, или **проецирует** (maps), виртуальные адреса на физические, по которым реально хранятся данные.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных системах теоретический максимум для общего виртуального адресного пространства составляет 4 Гб. По умолчанию Windows выделяет нижнюю половину этого пространства (в диапазоне адресов от x00000000 к x7FFFFFFF) процессам, а вторую половину (в диапазоне адресов от x80000000 к xFFFFFFFF) использует в своих целях

Виртуальная память процесса:

От 2 ГБ до 2 ТБ.

Кратна 64 КБ – гранулярность памяти пользовательского режима. Информацию о гранулярности можно получить с помощью GetSystemInfo().

Часть виртуальной памяти процесса, которая находится резидентно в физической памяти, называется рабочим набором – Working Set. Диапазон рабочего набора устанавливается функцией SetProcessWorkingSetSize(). Стандартный минимальный рабочий набор – 50 страниц по 4 КБ (200 КБ), стандартный максимальный рабочий набор – 345 страниц по 4 КБ (1380 КБ).

Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows.

Основные модули ОС Windows.

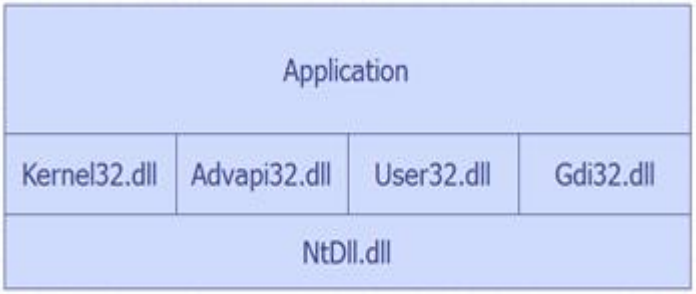
Пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как **Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll.**

Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:

1. функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows; Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб, который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле **Ntoskrnl.exe.**

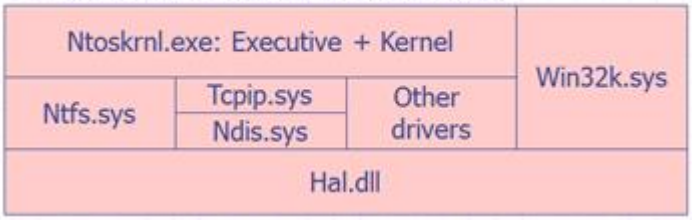
2. вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

- Архитектура приложения в пользовательском режиме (32-разрядная ОС):



  - Kernel32.dll – управление процессами, памятью, …
  - Advapi32.dll – управление реестром, безопасностью, …
  - User32.dll – управление окнами и сообщениями, …
  - Gdi32.dll – графический вывод.
  - NtDll.dll – интерфейсный модуль ядра.

- Архитектура системы в режиме ядра:



  - Ntoskrnl.exe (исполняющая система) – управление процессами и потоками, памятью, безопасностью, вводом-выводом, сетью, обменом данными.
  - Ntoskrnl.exe (ядро) – планирование потоков, обработка прерываний и исключений, реализация объектов ядра.
  - Ntfs.sys, Tcpip.sys, Ndis.sys, … – драйверы устройств.
  - Win32k.sys – реализация функций User32.dll и Gdi32.dll.
  - Hal.dll – интерфейсный модуль всей аппаратуры.

3. Системный реестр операционной системы Windows. Структура и главные разделы. Точки автозапуска программ. Средства редактирования реестра Windows. Функции работы с реестром из приложения.

Реестр Windows— иерархически построенная база данных параметров и настроек, состоящая из ульев.
В Windows элементы реестра хранятся в виде раздельных структур. Реестр подразделяется на составные части, которые разработчики этой операционной системы назвали ульями (hives) по аналогии с ячеистой структурой пчелиного улья. Улей представляет собой совокупность вложенных ключей и параметров, берущую начало в вершине иерархии реестра. Отличие ульев от других групп ключей состоит в том, что они являются постоянными компонентами реестра. Ульи не создаются динамически при загрузке операционной системы и не удаляются при ее остановке.
Реестр содержит данные, к которым Windows постоянно обращается во время загрузки, работы и её завершения, а именно:

- профили всех пользователей, то есть их настройки;
- конфигурация оборудования, установленного в операционной системе.
- данные об установленных программах и типах документов, создаваемых каждой программой;
- свойства папок и значков программ;
- данные об используемых портах.

Корневой раздел **HKCU** содержит данные, относящиеся к персональным настройкам и программной конфигурации локально вошедшего в систему пользователя. Он указывает на пользовательский профиль текущего вошедшего в систему пользователя, находящийся на жестком диске в файле \Users\<*имя_пользователя*>\Ntuser.dat
HKU Хранит информацию обо всех учетных записях, имеющихся на машине
HKCR Хранит файловые связи и информацию о регистрации объектов, относящихся к модели компонентных объектов — Component Object Model (COM)
HKLM Хранит информацию, связанную с системой
HKPD Хранит информацию о производительности
HKCC Хранит определенную информацию о текущем профиле оборудования

Внутреннее устройство реестра Кусты

На диске реестр не является обычным большим файлом, а представляет собой набор отдельных файлов, которые называются кустами. Каждый куст содержит дерево реестра, у которого есть раздел, служащий ему корнем или отправной точкой дерева. Подразделы и их параметры находятся ниже корня. Можно подумать, что корневые разделы, отображаемые в редакторе реестра, соответствуют корневым разделам в кустах, но так бывает не всегда. Путевые имена всех кустов, за исключением тех, которые используются для профилей пользователей, кодируются в диспетчере конфигурации. По мере того как диспетчер конфигурации загружает кусты, включая профили системы, он записывает путь к каждому кусту в параметрах подраздела HKLM\SYSTEM\CurrentControlSet\Control\Hivelist, удаляя путь при выгрузке куста. Он создает корневые разделы, связывает эти кусты вместе, чтобы построить структуру реестра, с которой вы знакомы и которая показывается редактором реестра.

Вы заметите, что некоторые из этих кустов могут изменяться и не имеют связанных с ними файлов. Система создает эти кусты и управляет ими целиком в памяти, поэтому такие кусты являются *временными*. *Система создает непостоянные кусты при каждой своей загрузке*. В качестве примера непостоянного куста можно привести HKLM\HARDWARE, в котором хранится информация о физических устройствах и выделенных этим устройствам ресурсах. Выделение ресурсов и определение установленного оборудования проводятся при каждой загрузке системы, поэтому хранить эти данные на диске было бы нелогично.

Первый блок куста называется базовым блоком. Базовый блок включает в себя глобальную информацию о кусте, в которую входят:
- сигнатура regf, которая идентифицирует файл как куст;
  - обновляемые последовательные номера;
  - отметка времени, показывающая, когда в последний раз в отношении куста применялась операция записи1;
  - информация о ремонте или восстановлении реестра, производимом Winload;
  - номер версии формата куста;
  - контрольная сумма;
- внутр. файловое имя файла куста (\Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM).

Windows упорядочивает данные реестра, которые куст хранит в контейнерах, называемых **ячейками**. В ячейке может храниться *раздел, параметр, дескриптор безопасности, список подразделов или список параметров раздела*. Четырехбайтовый символьный тег в начале данных ячейки описывает тип данных в виде сигнатуры.

Средства редактирования: regedit.exe, reg.exe.
regedit.exe – редактор реестра (в виде проводника).
reg.exe - редактирования системного реестра из командной строки.

Одна из многих возможностей реестра – это возможность задать **автозапуск программ** при старте ОС. Можно сделать автозапуск как для одного пользователя, так и для всех, при этом можно сделать чтоб программы запускались только один раз при входе в систему, после этого ключи программ автоматически удаляются из данного раздела реестра

Функции для работы с реестром

RegCloseKey Закрывает описатель ключа реестра.
RegCreateKeyEx Создает заданный ключ реестра.
RegDeleteKey/RegDeleteKeyEx Удаляет подключ и его значения
RegEnumKeyEx Перечисляет ключи заданного открытого ключа реестра.
RegEnumValue Перечисляет значения ключей заданного открытого ключа реестра.
RegGetValue Получает тип данных и сами данные значение ключа реестра.
RegLoadKey Создает подключ в HKEY_USERS или HKEY_LOCAL_MACHINE и сохраняет заданную информацию из файла в этот подключ.
RegOpenKeyEx Открывает заданный ключ реестра
RegSaveKey/RegSaveKeyEx Сохраняет заданный ключ и его подключи в файл.

4. Понятие окна в ОС Windows. Основные элементы окна. Понятие родительского и дочернего окна. Структура программы с событийным управлением. Минимальная программа для ОС Windows с окном на экране. Создание и отображение окна.

Понятие окна в ОС Windows.

Окно — графически выделенная часть экрана, принадлежащая какому-либо объекту, с которым работает пользователь. Окна могут иметь как произвольные, так и фиксированные (это характерно для диалоговых окон) размеры. Окно может занимать весь экран или только его часть. При этом на экране может быть одновременно выведено несколько (любое количество) окон.

Основные элементы окна. Понятие родительского и дочернего окна.

Каждое окно, создаваемое приложением, имеет **родительское окно**. При этом само оно по отношению к родительскому является дочерним. Какое окно является "основателем рода", т. е. родительским для всех остальных окон? Окна всех приложений располагаются в окне, представляющем собой поверхность рабочего стола Workplace Shell . Это окно, которое называется Desktop Window , создается автоматически при запуске операционной системы. Однако окно Desktop Window само по себе является дочерним по отношению к другому окну - окну Object Window . Это окно не отображается и используется системой Presentation Manager для собственных нужд. Родительское окно может иметь несколько **дочерних окон**, которые при этом называются окнами-братьями (или окнами-сестрами). У каждого дочернего окна может быть только одно родительское окно. Важной особенностью дочерних окон является то, что они всегда располагаются внутри своего родительского окна. Если пользователь попытается переместить дочернее окно за пределы родительского (например, при помощи мыши), будет нарисована только часть дочернего окна. Когда в одном родительском окне создано несколько дочерних окон, они могут перекрывать друг друга. Если пользователь перемещает родительское окно, то дочернее окно будет перемещаться вместе с ним. Когда пользователь изменяет размеры родительского окна, дочернее окно может отображаться не полностью. Если же пользователь минимизирует родительское окно, дочернее окно исчезает с поверхности экрана. При минимизации дочернего окна оно отображается в родительском окне в виде пиктограммы. При уничтожении родительского окна все его дочерние окна уничтожаются автоматически.

Структура программы с событийным управлением. Минимальная программа для ОС Windows с окном на экране

```
#include <windows.h>
int APIENTRY WinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
        WNDCLASSEX wcex; HWND hWnd; MSG msg;
        //заполнение wcex
...
        RegisterClassEx(&wcex);
        hWnd = CreateWindow("HelloWorldClass", "Hello, World!", WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
        ShowWindow(hWnd, nCmdShow);
        UpdateWindow(hWnd);
        while (GetMessage(&msg, NULL, 0, 0)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return (int)msg.wParam;
}
...
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message) {
                case WM_LBUTTONDBLCLK:
                        MessageBox(hWnd, "Hello, World!", "Message", MB_OK);
                        break;
                case WM_DESTROY:
                        PostQuitMessage(0);
```

```
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
  return 0;
}
```

Создание окна:

Функция CreateWindow:

```
HWND CreateWindow(
        LPCTSTR lpClassName,// Имя класса
        LPCTSTR lpWindowName,// Имя окна
        DWORD dwStyle,// Описывает стиль создаваемого окна
        int x,// Позиция по горизонтали верхнего левого угла окна в экранной системе координат.
        int y,// Позиция по вертикали верхнего левого угла окна в экранной системе координат.
        int nWidth,// Ширина окна в пикселях
        int nHeight,// Высота окна в пикселях.
        HWND hWndParent,// Дескриптор окна, которое является родителем данного.
        HMENU hMenu,// Дескриптор меню
        HANDLE hInstance,// Дескриптор экземпляра приложения с которым связано данное окно.
        LPVOID lpParam// указатель на определяемые пользователем данные
);
```

Функция CreateWindow возвращает дескриптор созданного ею окна (значение типа HWND). Если создать окно не удалось, значение дескриптора равно нулю.

Отображение окна:

Сперва мы вызываем функцию ShowWindow и передаем ей дескриптор только что созданного окна, чтобы Windows знала, какое окно должно быть показано. Мы также передаем число, определяющее в каком виде будет показано окно (обычным, свернутым, развернутым на весь экран и т.д.). После отображения окна мы должны обновить его. Это делает функция UpdateWindow; она получает один аргумент, являющийся дескриптором обновляемого окна.

```
ShowWindow(MainWindowHandle, show);
UpdateWindow(MainWindowHandle);
```

5. **Структура программы с событийным управлением. Структура события – оконного сообщения Windows. Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений. Процедуры посылки сообщений. Синхронные и асинхронные сообщения.**

Источники сообщений: Пользователь генерирует сообщения воздействуя на внешние устройства(мышь…); сама ОС посылает сообщения для уведомления ПО о событиях; Программа может вызывать функции ОС, результатом которой может являться посылка сообщения ПО; ПО может посылать сообщение самой себе; ПО может посылать сообщения другим прикладным программам
Оконная функция обрабатывает WM_DESTROY. Сообщение вызывается в результате функции DestroyWindow(). Эту функцию вызывает ОС. Если сообщение WM_DESTROY обрабатывается в главной оконной функции программы, то необходимо вызвать функцию:VOID PostQuitMessage(int exitCode), которая генерирует WM_QUIT.

Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений.

Создавая какой-либо поток, система предполагает, что он не будет иметь отношения к поддержке пользовательского интерфейса. Это позволяет уменьшить объем выделяемых ему системных ресурсов. Но, как только поток обратится к той или иной GUI-функции (например, для проверки очереди сообщений или создания окна), система автоматически выделит ему дополнительные ресурсы, необходимые для выполнения задач, связанных с пользовательским интерфейсом. А если конкретнее, то система создает структуру **THREADINFO** и сопоставляет ее с этим потоком. Элементы этой структуры используются, чтобы обмануть поток — заставить его считать, будто он выполняется в среде, принадлежащей только ему. THREADINFO — это внутренняя (недокументированная) структура, идентифицирующая *очередь асинхронных сообщений потока (posted-message queue), очередь синхронных сообщений потока (sent-message queue), очередь ответных сообщений (reply-message queue), очередь виртуального ввода (virtualized input queue) и флаги пробуждения (wake flags);*

События, поступающие от внешнего устройства, обрабатываются драйвером и помещаются в очередь. Далее они распределяются по приложениям. Для каждого приложения ОС организует прикладную очередь. В процессе распределения сообщений по прикладным очередям ОС извлекает очередное приложение системной очереди, определяет с каким окном связано это сообщение, помещает это сообщение в очередь того приложения, которому принадлежит окно. Часто говорят, что сообщения передаются окнам и обрабатываются окнами.

Синхронными сообщениями называются сообщения, которые Windows помещает в очередь сообщений приложения. Такие сообщения извлекаются и диспетчеризуются в цикле обработки сообщений. , к ним относятся сообщения о событиях пользовательского ввода, таких как нажатие клавиш (WM_KEYDOWN и WM_KEYUP), перемещение мыши (WM_M0USEM0VE) или щелчок левой кнопкой мыши (WM_LBUTT0ND0WN).

*LRESULT **SendMessage**(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);*

Оконная процедура обработает сообщение, ***и только по окончании обработки функция SendMessage вернет управление.***

Асинхронные сообщения передаются непосредственно окну, когда Windows вызывает оконную процедуру. Остальные сообщения, как правило, являются асинхронными. Приложение также может послать асинхронное сообщение, вызвав функцию SendMessage.

*BOOL **PostMessage**(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);*

При вызове этой функции система определяет, каким потоком создано окно, идентифицируемое параметром hwnd. Далее система выделяет блок памяти, сохраняет в нем параметры сообщения и записывает этот блок в очередь асинхронных сообщений данного потока. Кроме того, функция устанавливает флаг пробуждения QS_POSTMESSAGE (о нем — чуть позже). Возврат из PostMessage происходит сразу после того, как сообщение поставлено в очередь, поэтому вызывающий поток остается в неведении, обработано ли оно процедурой соответствующего окна. На самом деле вполне вероятно, что окно даже не получит это сообщение. Такое возможно, если поток, создавший это окно, завершится до того, как обработает все сообщения из своей очереди.

Сообщение можно поставить в очередь асинхронных сообщений потока и вызовом PostThreadMessage:

*BOOL **PostThreadMessage**(DWORD dwThreadId, UINT uMsg, WPARAM wParam, LPARAM lParam);*

Какой поток создал окно, можно определить с помощью GetWindowThreadProcessId:

*DWORD **GetWindowThreadProcessId**(HWND hwnd,PDWORD pdwProcessId);*

*VOID **PostQuitMessage**(int nExitCode);*

Она вызывается для того, чтобы завершить цикл выборки сообщений потока. Не помещает сообщение ни в одну из очередей структуры THREADINFO. Эта функция просто устанавливает флаг пробуждения QS_QUIT и элемент nExitCode структуры THREADINFO. Так как эти операции не могут вызвать ошибку, функция PostQuitMessage не возвращает никаких значений (VOID).

1) LRESULT ***SendMessageTimeout****(HWND hwnd, UINT uMsg,WPARAM wParam, LPARAM lParam, UINT fuFlags, UINT uTimeout, PDWORD_PTR pdwResult);*

Она позволяет задавать отрезок времени, в течение которого Вы готовы ждать ответа от другого потока на Ваше сообщение.

2)BOOL ***SendMessageCallback****(HWND hwnd, UINT uMsg, WPARAM wParam,LPARAM lParam, SENDASYNCPROC pfnResultCallBack, ULONG_PTR dwData);*

И вновь первые четыре параметра идентичны параметрам функции SendMessage. При вызове Вашим потоком SendMessageCallback отправляет сообщение в очередь синхронных сообщений потока-приемника и тут же возвращает управление вызывающему (т. е. Вашему) потоку. *Закончив обработку сообщения, поток-приемник асинхронно отправляет свое сообщение в очередь ответных сообщений Вашего потока.* Позже система уведомит Ваш поток об этом, вызвав написанную Вами функцию; у нее должен быть следующий прототип:

VOID CALLBACK ***ResultCallBack****(HWND hwnd, UINT uMsg, ULONG_PTR dwData, LRESULT lResult);*

3) BOOL ***SendNotifyMessage****(HWND hwnd, UINT uMsg, WPARAM wParam,LPARAM lParam);*

Поместив сообщение в очередь синхронных сообщений потока-приемника, она немедленно возвращает управление вызывающему потоку. Если SendNotifyMessage посылает сообщение окну, созданному другим потоком, приоритет данного синхронного сообщения выше приоритета асинхронных сообщений, находящихся в очереди потока-приемника. Иными словами, сообщения, помещаемые в очередь с помощью SendNotifyMessage, всегда извлекаются до выборки сообщений, отправленных через PostMessage. Если сообщение посылается окну, созданному вызывающим потоком, SendNotifyMessage работает точно так же, как и SendMessage, т. е. не возвращает управление до окончания обработки сообщения.

4) BOOL ***ReplyMessage****(LRESULT lResult);* *вызывается потоком, принимающим оконное сообщение*. Вызвав ее, поток как бы говорит системе, что он уже получил результат обработки сообщения и что этот результат нужно упаковать и асинхронно отправить в очередь ответных сообщений потока-отправителя. Последний сможет пробудиться, получить результат и возобновить работу.

6. **Ввод данных с манипулятора «мышь». Обработка сообщений мыши. Ввод данных с клавиатуры. Понятие фокуса ввода. Обработка сообщений от клавиатуры.**

При каждом нажатии и отпускании левой кнопки мыши ОС посылает программе сообщение WM_LBUTTONDOWN и WM_LBUTTONUP. При перемещении мыши ОС помещает в очередь сообщение WM_MOUSEMOVE. Если в оконном классе окна над которым находится указатель мыши установлен стиль CS_DBLCLKS, то окно способно получать сообщение двукратного щелчка. Когда двукратные щелчки разрешены, ОС устанавливает один из внутренних таймеров заданный в CotrolPanel. Если пользователь в пределах этого интервала совершает двойное нажатие, то ОС вместо сообщения о нажатии посылает сообщение двукратного щелчка WM_LBUTTONDBLCLK.В сообщении от мыши параметры имеют следующий смысл: WPARAM – определяет состояние кнопок мыши и клавиш Ctrl & Shift. LPARAM – младшие 2 байта кодируют координату X, старшие – Y.

При нажатии и отпускании обычной клавиши ОС генерирует сообщение WM_KEYDOWN & WM_KEYUP. Если нажаты системные клавиши WM_SYSKEYDOWN & WM_SYSKEYUP (Они соответствуют системному нажатию и отпусканию. Системное нажатие происходит с клавишей Alt !!!). Если пользователь нажал клавишу и удерживает ее, То происходит автоповтор клавиатуры. ОС автоматически начинает помещать в очередь сообщений WM_KEYUP. В результате на 1 сообщение WM_KEYDOWN может быть несколько WM_KEYUP. Если очередь сообщений забивается, то в параметре сообщений WM_KEYDOWN ОС начинает увеличивать счетчик повторов. wParam – код виртуальный. Для символьных сообщений там символьный код. При отпускании клавиши TranslateMessage помещает в очередь сообщений WM_DEADCHAR (WM_SYSDEADCHAR). lParam во всех типах – набор битовых флагов: Счетчик повтора; Индикатор расширенной клавиши; Индикатор системной клавиши (удерживался ли Alt); Индикатор предыдущего состояния, который показывает - была ли до этого нажата эта клавиша; Индикатор текущего состояния

Параметры клавиатурных сообщений не несут информацию о состоянии Ctrl & Shift. Чтобы ее получить нужно вызвать функцию GetKeyState(UINT virtkKey). Она принимает код клавиши и возвращает ее состояние нажата/отпущена (включена/выключена). Эта функция синхронизирована с моментом посылки последнего клавиатурного сообщения. Т.е. она возвращает состояние клавиши не на момент ее вызова, а на момент последнего сообщения от клавиатуры. GetAsyncKeyState – позволяет определить состояние клавиатуры на момент вызова функции.

Фокусом ввода всегда владеет либо активное окно, либо одно из его дочерних окон. Часто дочерними окнами являются элементы управления — кнопки, переключатели, флажки, текстовые поля и списки, которые обычно размещаются в окне диалога. Обрабатывая сообщения WNLSETFOCUS и WM_KILLFOCUS, оконная процедура может определить текущий статус связанного с ней окна. Первое сообщение показывает, что окно получило фокус ввода, второе — что окно потеряло фокус ввода.Для работы с фокусом ввода предусмотрены следующие функции:

Функция HWND SetFocus(HWND hWnd) устанавливает фокус ввода на окно hWnd, возвращая дескриптор окна, которое располагало фокусом до вызова функции.Функция HWND GetFocus() возвращает дескриптор окна, имеющего фокус ввода в текущий момент.

7. **Вывод информации в окно. Механизм перерисовки окна. Понятие области обновления окна. Операции с областью обновления окна.**

Вывод в окно

Разделение дисплея между прикладными программами осуществляется с помощью окон. Видимая площадь окна может изменяться, что требует постоянного контроля за отображаемой в окне информацией и своевременного восстановления утраченных частей изображения. ОС не хранит графическую копию рабочей (пользовательской) части каждого окна. Она возлагает ответственность за правильное отображение окна на прикладную программу, посылая ей WM_PAINT каждый раз, когда все окно или его часть требует перерисовки.

При операциях с окнами система помечает разрушенные части окна, как подлежащие обновлению и помещает информацию о них, в специальную область: **область обновления** – UPDATEREGION. На основании содержимого этой области и происходит восстановление. ОС посылает окну сообщение WM_PAINT всякий раз, когда область обновления окна оказывается не пустой и при условии, что в очереди сообщений приложения нет ни одного сообщения. При получении сообщения WM_PAINT, окно должно перерисовать лишь свою внутреннюю часть, называемую рабочей областью(ClientArea). Все остальные области окна перерисовывает ОС по WM_NCPAINT.

Для ускорения графического вывода Windows осуществляет отсечение. На экране перерисовываются лишь те области окна, которые действительно требуют обновления. Вывод за границами области отсечения игнорируется. Это дает право прикладной программе перерисовывать всю рабочую область в ответ на сообщение WM_PAINT. Лишний вывод ОС отсекает.

Инициатором сообщения WM_PAINTможет выступать не только ОС, но и прикладная программа. Чтобы спровоцировать перерисовку окна необходимо вызвать функцию:

void InvalidateRect(HWND, //handleокна
RECT* //эта область требует перерисовки,
BOOL//нужно ли перед перерисовкой очищать область обновления);

Очистка производится сообщением WM_ERASE_BACKGROUND.

После вызова функции InvalidateRect, окно не перерисовывается сразу (до WM_PAINT). Перерисовка произойдет только при опросе программой очереди сообщений. Когда перерисовка требуется немедленно, то вслед за InvalidateRect вызывается функция: void UpdateWindow(HWND);

Перерисовка окна

Перерисовка содержимого окна основана на получении контекста устройства, связанного с окном, рисование (вывод графических примитивов) в этом контексте устройства, и освобождение контекста устройства.

В разных случаях получение контекста устройства осуществляется разными функциями ОС. В ответ на сообщение WM_PAINTконтекст устройства получается с помощью функции:

HDCBeginPaint(HWND,PAINTSTRUCT*);
//рисование
voidEndPaint(HWND,PAINTSTRUCT*);

Между вызовами этих 2х функций заключаются вызовы графических примитивов (Rectangle(),Line()).

Функции BeginPainиEndPaintможно вызывать только на сообщениеWM_PAINT.

Иногда бывает необходимо выполнить перерисовку окна в какой-то другой момент времени (по сообщению от таймера). В этом случае контекст дисплея получается с помощью функции:

HDC GetDC(HWND);

А освобождается функцией:

int ReleaseDC(HWND, HDC);

Вызовы этих функций обязательно должны быть сбалансированы, иначе возникнут сбои в работе ОС.

В Windows передача сообщения в окно всегда осуществляется синхронно: отправитель не может продолжить работу, пока окно не обработает полученное сообщение.

8. **Принципы построения графической подсистемы ОС Windows. Понятие контекста устройства. Вывод графической информации на физическое устройство. Управление цветом. Палитры цветов. Графические инструменты. Рисование геометрических фигур.**

Подмножество Функций ОС Windows для вывода графической информации на экран и другие внешние устройства называется GDI (Graphic Device Interface). Принципы: 1)GDI - аппаратно независим (работаем с виртуальным устройством). Качество выводимого изображения определяется физическими свойствами адаптера. 2)Зависимость от устройств отображения достигается за счет использования драйверов. При смене драйверов только меняем драйвер и программа работает нормально 3)Все элементы графического изображения описываются в рамках логической системы координат[1], которая может отличаться от физической. На экране изображение является плоским. Значения координат по обоим осям изменяется в пределах (зависит от ОС) (-32768..32767).4)3 основные цвета: RGB. Каждому из цветов отводится по 1 байту.5)GDI позволяет строить изображение по принципу WYSIWYG (What You See Is What You Get). Это обеспечивается не только применение логической системы координат, но и масштабируемых шрифтов TrueType. ***КОНТЕКСТ УСТРОЙСТВА*** - логический объект ОС, который связан с физическим устройством и заменяет его в функциях вывода. Структура DC не доступна, но доступны функции создающие/получающие дескриптор контекста устройства по каким-то входным параметрам. Дескриптор передается первым параметром в функцию осуществляющую вывод графического примитива. Специальные функции вывода позволяют интерпретировать каждое окно на экране как отдельное устройство. Приложение которое запрашивает контекст устройства для конкретного окна, получает DC внутри этого окна и не может осуществлять доступ за его пределы. Win32 API поддерживает следующие типы контекстов устройства: контекст дисплея; контекст принтера; контекст в памяти (совместимый контекст); метафайловый контекст; информационный контекст.

ВЫВОД ИНФОРМАЦИИ Вывод изображений на такое устройство, как принтер, может выполняться с использованием тех же приемов, что и вывод в окно приложения. Прежде всего необходимо получить контекст устройства. Затем можно вызывать функции GDI, выполняющие рисование, передавая им идентификатор полученного контекста в качестве параметра.

В отличие от контекста отображения, контекст физического устройства не получается, а создается, для чего используется функция CreateDC :

```
HDC WINAPI CreateDC(
LPCSTR lpszDriver, // имя драйвера
LPCSTR lpszDevice, // имя устройства
LPCSTR lpszOutput, // имя файла или порта вывода
const void FAR* lpvInitData); // данные для инициализации
```

Рисование геометрических фигур

```
BOOL LineTo( HDC hdc, int nXEnd, int nYEnd );
BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint/*old current position*/);
BOOL Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect );
BOOL Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);
BOOL Polygon(HDC hdc, // handle to DC CONST POINT *lpPoints, // polygon vertices int nCount // count of polygon vertices);
BOOL PolyBezier(HDC hdc, // handle to device context CONST POINT* lppt, // endpoints and control points
DWORD cPoints // count of endpoints and control points);
```

Графические инструменты

Pen – CreatePen, Brush – CreateSolidBrush, Font – CreateFont, Холст – CreateCompatibleDC (?)

Управление цветом.

RGB –формат. Для кодирования цвета используются переменные с типом данных COLORREF, который определен через тип данных UINT.

```
COLORREF col; Col = RGB(255,0,0); // в памяти по байтам: 0,B,G,R.
BYTE RedValue; RedValue=GetRValue(color) //значения составляющих GetGValue, GetBValue
```

Позволяет иметь более 16 миллионов оттенков. Далеко не все графические устройства поддерживают такое количество. Если программа устанавливает цвет, который данное устройство воспроизвести не может ОС заменяет этот цвет на ближайший из числа доступных.

```
COLOREF GetNearestColor(HDC, COLORREF).
```

Палитры цветов.

Для того чтобы создать палитру, приложение должно заполнить структуру LOGPALETTE , описывающую палитру, и массив структур PALETTEENTRY , определяющий содержимое палитры.

Структура LOGPALETTE и указатели на нее определены в файле windows.h:

```
typedef struct tagLOGPALETTE
{
WORD palVersion;
WORD palNumEntries;
PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

В поле palNumEntries нужно записать размер палитры (количество элементов в массиве структур PALETTEENTRY).

Сразу после структуры LOGPALETTE в памяти должен следовать массив структур PALETTEENTRY, описывающих содержимое палитры:

```
typedef struct tagPALETTEENTRY
{
BYTE peRed;
BYTE peGreen;
BYTE peBlue;
BYTE peFlags;
} PALETTEENTRY;
typedef PALETTEENTRY FAR* LPPALETTEENTRY;
```

9. Растровые изображения. Виды растровых изображений. Значки и курсоры. Способ вывода растровых изображений с эффектом прозрачного фона. Аппаратно-зависимые и аппаратно-независимые растровые изображения. Операции с растровыми изображениями. Вывод растровых изображений.

Типы растровых изображений:
Bitmap – базовый формат растрового изображения.
Icon – значок: AND-маска и XOR-маска.
Cursor – курсор: две маски и точка касания – Hot Spot.
Значки – это небольшая картинка, ассоциируемая с некоторой программой, файлом на экране. Значок является частным случаем растровых изображений. На экране значки могут иметь не прямоугольную форму, что достигается за счет описания значка двумя точечными рисунками: AND-mask. Монохромная. XOR-mask. Цветная.
При выводе значков, ОС комбинирует маски по следующему правилу:
Экран = (Экран ANDМонохромная маска)XORЦветная маска.
Накладывая AND-mask, ОС вырезает на экране пустую область с заданным контуром. BAND-maskфигура кодируется с помощью 0, а прозрачный фон с помощью 1.
После вывода AND-maskОС накладываетXOR-mask, содержащую изображения фигур. Изображение фигуры является цветным.
На диске значки сохраняются в *.ico формате. В ОС существует несколько форматов значков, которые отличаются по размеру и цвету (16x16, 32x32, 16x32, 64x64).
Курсоры. Указатели мыши. Небольшой образ. По своему представлению в файле и памяти курсор напоминает значки, но существуют некоторые значки. Курсоры могут быть размером 16x16 и 32x32. Важным существенным отличием является наличие в нем горячей точки (hotspot), которая ассоциируется с позицией указателя мыши на экране. *.CUR.
Точечные рисунки – это изображение, представление графической информации, ориентированное на матричное устройство вывода. Точечный рисунок состоит из пикселей, организованных в матрицу. ОС позволяет использовать точечные рисунки двух видов:
Аппаратно-зависимые. Device Dependent Bitmpap. Рассчитаны только на определенный тип графического адаптера или принтера. Их точки находятся в прямом соответствии с пикселями экрана или другой поверхности отображения.
Если это экран – то информация о пикселях представляется битовыми планами в соответствии с особенностями устройства. Он наименее удобен при операциях с точечным рисунком, но обеспечивает наибольшую скорость графического вывода. Он хорошо подходит для работы с точечными рисунками в оперативной памяти.
При хранении на диске используется аппаратно-независимый формат - BMP,DIB.
Аппаратно-независимые. Device Independent Bitmpap. Формат хранения аппаратно-независимых точечных рисунков не зависит от используемой аппаратуры. Здесь информация о цвете и самом изображении хранится раздельно.
Цвета собраны в таблицу, а точки изображения кодируют номера цветов таблицы. Под каждую точку изображения может отводиться 1,4,8,16,24 битов изображения. Они могут храниться на диске в сжатом виде.
Для сжатия применяется алгоритм RunLengthEncoding(RLE). Разжатие производится автоматически. Недостаток: обеспечивается более низкая скорость работы.
Вывод растрового изображения с эффектом прозрачного фона: AND-маска – монохромная. Фигура кодируется нулем, прозрачный фон – единицей. Вырезает на экране «черную дыру» там, где должна быть фигура.
Растровая операция – способ комбинирования пикселей исходного изображения с пикселями поверхности отображения целевого контекста устройства. При масштабировании в сторону сжатия некоторые цвета могут пропадать. При растяжении, таких проблем не существует. При сжатии возможно 3 способа сжатия.

10. Библиотека работы с двумерной графикой Direct2D. Инициализация библиотеки. Фабрика графических объектов библиотеки Direct2D. Вывод графики средствами библиотеки Direct2D. (потом переведу)

Для удовлетворения новых потребностей рынка IT-техноолгий корпорация Microsoft летом 2009 года разработала на базе технологии DirectX 10 набор библиотек для работы и вывода двумерной графики — Direct2D.

Direct2D включает в себя 4 заголовочных файла и одну библиотеку:

. d2d1.h — содержит объявления основных функций Direct2D API на языке C и C++;
. d2d1helper.h — содержит вспомогательный структуры, классы, функции;
. d2dbasetypes.h — определяет основные примитивы Direct2D, включен в d2d1.h;
. d2derr.h — определяет коды ошибок Direct2D. Включен в d2d1.h;
. d2d1.lib — двоичная библиотека, содержащая все объявленные в заголовочных файлах функции.

Как и DirectX, Direct2D построен на модели COM. Основным объектом, который предоставляет интерфейсы для создания других объектов, является Фабрика Direct2D, или объект класса ID2D1Factory. Он имеет в своем составе методы типа CreateResourse, которые позволяют создавать объекты более специфических типов.

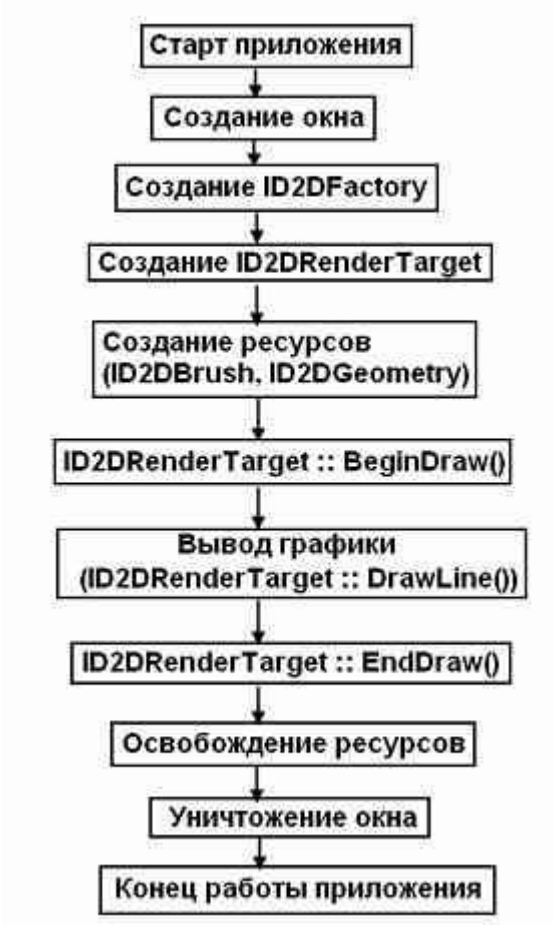
Все объекты (ресурсы) в Direct2D делятся на два больших типа — устройство-зависимые (device-dependent) и устройство-независимые (device-independent). Устройство-зависимые объекты ассоциируются с конкретным устройством вывода и должны быть реинициализированы каждый раз, когда устройство, с которым они ассоциируются, требует реинициализации. Устройство-независимые ресурсы существуют без привязки к какому-либо устройству и уничтожаются в конце жизненного цикла программы или по желанию программиста. Классификация и основные примеры ресурсов приведены на рисунке 1.

Объекты класса ID2DRenderTarget — это устройство-зависимые объекты, которые ассоциируются с конкретным устройством вывода. Это может быть конкретное окно приложения, битовый образ (изображение) или другое устройство. ID2DRenderTarget имеет в своем составе методы BeginDraw() и EndDraw(), между которыми выполняются все операции вывода графической информации на устройство вывода.

В качестве инструмента вывода используются объекты класса ID2DBrush, которые задают цвет и другие парамеиры выводимых объектов (в т.ч. градиентные заливки). И ID2DBrush, и ID2DRenderTarget — устройство-зависимые ресурсы и будучи созданными однажды для конкретного устройства, могут применяться лишь к нему, и должны быть уничтожены всякий раз, когда уничтожается их устройство.

Объект класса ID2DGeometry — устройство-независимый ресурс. Будучи созданным однажды, он может быть использован любым объектов ID2DRenderTarget. ID2DGeometry задает двумерную форму, интерполированную треугольниками.

После того, как работа с ресурсами и объектами завершена, они должны быть уничтожены функцией Release(), которая унаследована ими от базового COM-объекта. При этом по возможности стоит избегать частого создания и освобождения ресурсов, так как этот процесс требует достаточно много ресурсов процессора. Общая схема работы с Direct2D:



11. Вывод текста в ОС Windows. Понятие шрифта. Характеристики шрифта. Понятия физического и логического шрифта. Операции с физическими шрифтами. Операции с логическими шрифтами. Параметры ширины и высоты логического шрифта.

1) BOOL TextOut Функция TextOut записывает строку символов в заданном месте, используя текущий выбранный шрифт, цвет фона и цвет текста.
Если функция завершается с ошибкой, величина возвращаемого значения – ноль, иначе – не ноль.
SetTextAlign - устанавливает флажки выравнивания текста для заданного контекста устройства.
GetTextAlign - извлекает настройки выравнивания текста для заданного контекста устройства.
SetTextColor - function sets the text color for the specified device context to the specified color.
GetTextColor - function retrieves the current text color for the specified device context.
2) BOOL ExtTextOut
Выводит текст используя текущий выбранный шрифт, цвета фона и текста. Если строка рисуется, возвращаемое значение является отличным от нуля.
3) BOOL PolyTextOut
рисует несколько строк, используя шрифт и цвета текста, в настоящее время выбранные в заданном контексте устройства. При ошибке – 0, иначе – не ноль.
4)LONG TabbedTextOut
Пишет строку символов в заданном месте, разворачивая позиции табуляции в значения, указанные в массиве позиций табуляции. Текст пишется в текущем выбранном шрифте, цвете фона и цвете текста.
Если функция завершается ошибкой, возвращаемое значение – нуль.
5) int DrawText
Рисует отформатированный текст в заданном прямоугольнике. Если функция завершается успешно, возвращаемое значение - высота текста в логических единицах измерения.
6) int DrawTextEx
Рисует форматированный текст в заданном прямоугольнике.Если функция завершается с ошибкой, величина возвращаемого значения - ноль.
Шрифт – множество символов со сходными размерами и начертанием контуров. Семейство шрифта – набор шрифтов со сходной шириной символов и гарнитурой:
Физические – устанавливаемые в операционную систему, файлы.
Логические – запрашиваемые программой у операционной системы, LOGFONT.
Физический шрифт Установка шрифта:
Скопировать файл шрифта в C:\Windows\Fonts.
Вызвать int AddFontResource(LPCTSTR lpszFilename).
Вызвать SendMessage с кодом WM_FONTCHANGE.
Удаление шрифта:
Вызвать bool RemoveFontResource(LPCTSTR lpszFilename).
Удалить файл шрифта из C:\Windows\Fonts.
Вызвать SendMessage с кодом WM_FONTCHANGE.
Логический шрифт Создание логического шрифта (LOGFONT):
CreateFontIndirect / CreateFont,
SelectObject DeleteObject
Параметры:
- гарнитура (typeface): с засечками, без засечек
- Начертание (style) – полужирный, курсив.
- Кегль (size) – размер в пунктах, 10 pt = 3.76 мм.

12. Системы координат. Трансформации. Матрица трансформаций. Виды трансформаций и их представление в матрице трансформаций. Преобразования в страничной системе координат. Режимы масштабирования.

Мировая – world coordinate space (2^32). Обеспечивает параллельный перенос, масштабирование, отражение, поворот, наклон.
Логическая (страничная) – page coordinate space (2^32). Устаревшая система координат, основанная на режимах масштабирования (mapping modes). Обеспечивает параллельный перенос, масштабирование, отражение.
Устройства – device coordinate space (2^27). Обеспечивает параллельный перенос (к началу координат на устройстве).
Физическая – physical device coordinate space. Например, клиентская область окна на экране.

Трансформации

Включить расширенный графический режим: int SetGraphicsMode(HDC hdc, int iMode); GM_ADVANCED
При пересчете Windows осуществляет пересчет логической точки (LP) из логического пространства координат, в физическую точку из физической системы координат (DP). Это делается за 3 шага:
1. Параллельный перенос изображения на логической плоскости путем вычитания из координат каждой точки изображения заданных константных значений.
2. Масштабирование полученного изображения путем масштабирования заданной точки (умножением на заданный коэффициент). Изображение переносится на физическую плоскость.
3. Параллельный перенос изображения на физической плоскости за счет добавления заданных константных значений.

$D_X=(L_X-XWO)*XVE/XWE+XVO \quad D_Y=(L_Y-YWO)*YVE/YWE+YVO$

Где: L_X – координата X в логической системе XWO – смещение по оси X в логической системе
XVO – смещение по оси X в физической системе координат XVE/XWE – масштабный интерфейс по оси X
В ОС существуют функции, которые выполняют заданные преобразования для массива точек: **LPtoDP()** и **DPtoLP()**.

Матрица трансформации:

Struct XFORM {FLOAT eM11,eM12,eM21,eM22, eDx,eDy}
|eM11 eM12 0|
|eM21 eM22 0|
|eDx eDy 1| (ris_1)
Применение матрицы трансформации:
|x` y` 1| = |x y 1| * (ris_1)

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

x' = x * eM11 + y * eM21 + eDx
y' = x * eM12 + y * eM22 + eDy
a - Изменение масштаба по горизонтали. Значение больше 1 расширяет элемент, меньше 1, наоборот, сжимает.
b - Наклон по горизонтали. Положительное значение наклоняет влево, отрицательное вправо.
c - Наклон по вертикали. Положительное значение наклоняет вверх, отрицательное вниз.
d - Изменение масштаба по вертикали. Значение больше 1 расширяет элемент, меньше 1 — сжимает.
tx - Смещение по горизонтали в пикселах. Положительное значение сдвигает элемент вправо на заданное число пикселов, отрицательное значение сдвигает влево.
ty - Смещение по вертикали в пикселах. При положительном значении элемент опускается на заданное число пикселов вниз или вверх при отрицательном значении.

Виды трансформаций

Параллельный перенос: $|x'\ y'\ 1| = |x\ y\ 1| * \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ eDx & eDy & 1 \end{vmatrix}$

Масштабирование: $|x'\ y'\ 1| = |x\ y\ 1| * \begin{vmatrix} eM11 & 0 & 0 \\ 0 & eM22 & 0 \\ 0 & 0 & 1 \end{vmatrix}$

Отражение: $|x'\ y'\ 1| = |x\ y\ 1| * \begin{vmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$

Поворот $|x'\ y'\ 1| = |\ x\ \ y\ 1| * \begin{vmatrix} cos & sin & 0 \\ -sin & cos & 0 \\ 0 & 0 & 1 \end{vmatrix}$ Наклон: $|x'\ y'\ 1| = |\ x\ \ y\ 1| * \begin{vmatrix} 1 & eM12 & 0 \\ eM21 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$

Режимы масштабирования

В процессе вывода изображения функции графического интерфейса GDI преобразуют логические координаты в физические. Для определения способа такого преобразования используется атрибут с названием режим масштабирования (mapping mode), который хранится в контексте устройства вывода.

Для установки типа масштабирования используется метод контекста устройства **int SetMapMode(HDC hdc, int fnMapMode)**, а для получения типа масштабирования - метод **GetMapMode()**.

Для указания режима масштабирования в файле windows.h определены символьные константы с префиксом MM_ (от Mapping Mode - режим масштабирования).

Восемь существующих режимов масштабирования координат задаются с помощью символьных констант, определенных в файле Wingdi.h:

13. Понятие ресурсов программ Windows. Виды ресурсов. Операции с ресурсами.

Ресурсы – двоичные данные, записываемые в исполняемый модуль приложения. Стандартные типы ресурсов:

Курсор – Cursor,Картинка – Bitmap,Значок – Icon,Меню – Menu,Окно диалога – Dialog Box,Таблица строк – String Table,Таблица сообщений (об ошибках) – Message Table,Шрифт – Font,Таблица горячих клавиш – Accelerator Table

Ресурсы могут быть разделяемыми, когда несколько процессов могут их использовать одновременно или параллельно, и неделимыми.

Ресурс может быть выделен задаче, в следующих случаях:1.) ресурс свободен, и в системе нет запросов от задач более высокого приоритета к запрашиваемому ресурсу;2.) текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;3.) ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Добавление и удаление ресурсов исполняемого модуля:

HANDLE BeginUpdateResource(LPCTSTR pFileName, bool bDeleteExistingResources);

bool UpdateResource(HANDLE hUpdate, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage, void* lpData, DWORD cbData);

bool EndUpdateResource(HANDLE hUpdate, bool fDiscard);

Загрузка ресурсов из исполняемого модуля:

HRSRC FindResourceEx(HMODULE hModule, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage); FindResource, EnumResourceXxx.

HGLOBAL LoadResource(HMODULE hModule, HRSRC hResInfo); LoadImage, LoadMenu, LoadXxx.

DWORD SizeofResource(HMODULE hModule, HRSRC hResInfo);

Процесс записи ресурсов в файл начинается с вызова BeginUpdateResource. При этом флаг bDeleteExistingResources задает режим записи: с удалением существующих ресурсов или без.Заканчивается процесс записи вызовом EndUpdateResource. Если флаг bDiscard установлен в TRUE, то запись ресурсов отменяется, в противном случае ресурсы записываются в файл.

Между вызовами этих двух функций можно обновлять ресурсы с помощью функции UpdateResource, причем вызывать ее можно неоднократно.Функция UpdateResource добавляет, удаляет или заменяет ресурс в исполняемом файле.

Функция FindResource выясняет место ресурса с заданным типом и именем в указанном модуле.

FindResource, поиск любого типа ресурса, но эта функция должна использоваться, только в том случае, если прикладная программа должна получить доступ к двоичным данным ресурса при производстве последующих вызовов функции LockResource. Функция LoadResource загружает указанный ресурс в глобальную память.

Файл ресурсов описывает и диалоговые окна. Все компоненты окна - кнопки, элементы управления и даже статический текст, имеют свои идентификаторы, координаты в окне и номера, как имеют их и элементы меню. Для каждого окна нужная своя функция окна, поэтому в программу помимо WndProc - функции главного окна, придётся включить PifProc() - функцию диалогового окна. Аргументы у неё будут такие же, как и у WndProc.

14. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом статического импорта процедур. Соглашения о вызовах процедур DLL-библиотеки. Точка входа-выхода DLL-библиотеки.

Динамически-загружаемая библиотека (DLL) – двоичный модуль операционной системы. Это программа с множеством точек входа. Включает код, данные и ресурсы.
Подключение DLL называется **импортом.** DLL представляет собой дополняемый модуль ОС (обычно DLL), код и ресурсы которого могут использоваться в составе других динамических библиотек и приложений. DLL – программа с множеством точек входа. В отличие от статической библиотеки, которая включается в выполняемый exe модуль на этапе сборки, динамическая библиотека собирается в виде отдельно выполняемого модуля. Использование динамической библиотеки выполняется одним из двух способов:
- Статического импорта
- Динамического импорта

Для создания DLL в разных языках используются разные средства. В C++:
Создание: __declspec(dllexport) int Min(int X, int Y);
При статическом импорте подключение DLL осуществляется наподобие обычных библиотек. Разница лишь в описании внешней функции. В C++: __declspec(dllimport) int Min(int X, int Y). Такого описания функции в исходном тексте функции не достаточно. Система требует подключения так называемой библиотеки импорта (lib-файла) при компоновке программы. Библиотека импорта создается системой Visual C автоматически при компиляции DLL. При подключении DLL необходимо знать еще один важный параметр – соглашение о вызове подпрограмм. Существуют следующие соглашения о вызове подпрограмм в ОС Windows:
- __cdecl. Параметры передаются на стек в обратном порядке. За освобождение стека после вызова подпрограммы отвечает вызывающая программа.
- __pascal. Передача параметров на стек в прямом порядке. Освобождение стека осуществляет сама вызванная подпрограмма.
- __stdcall. Соглашение для стандартных DLL Windows. Передача параметров на стек происходит в обратном порядке. Освобождение стека выполняет вызванная подпрограмма.
- __register. Передача параметров преимущественно через регистры процессора. Это не используется при создании DLL (это не стандартизировано).

Соглашение о вызове должно записываться в прототипе функции.
__declspec(dllexport) int __cdecl Min(int X, int Y);
Функция в DLL получает имя: _<имя_функции>@<количество_байт_параметров>. _Min@8.
Существует еще один способ создания библиотеки import. Библиотека импорта может создаваться на основе существующей DLL библиотеки. Она создается непосредственно из файла описания DLL библиотеки. Файл описания DLL имеет расширение DEF, является текстовым файлом, в котором перечислены имена функций экспортируемых из DLL. Справа от имени функции через разделитель записывается номер функции в DLL.
EXPORTS _Min@8 @1 _Max@8 @2

Этот номер может использоваться для импорта функции. Лучше никогда не использовать вызов по номерам. Рекомендуется создавать DEF-файл вручную и включать его в проект VC++. Компилятор умеет обнаруживать в проекте DEF-файл и использовать его для именования функций в DLL. DEF-файл, который необходимо включать в проект, и DEF-файл передаваемый пользователю DLL (заказчику) отличаются. Первый файл записывается без _ & @8, т.е. Min.

ФУНКЦИЯ ВХОДА*/ВЫХОДА*

DLL может иметь НЕОБЯЗАТЕЛЬНУЮ функцию BOOL WINAPI DllMain(HINSTANCE hInst, DWORD dwReason, LPVOID lpImpload), которая вызывается системой Windows автоматически в 4х случаях. hInst – дескриптор загружаемого модуля, который равен адресу, с которого DLL проецируется в память. dwReason – причина вызова функции (одна из 4х причин). lpImpload - показывает, как DLL загружается в память (методом неявной загрузки – статический импорт, или методом явной загрузки – динамический импорт). 4 причины вызова:
1. DLL_PROCESS_ATTACH – при первой загрузке DLL каким-либо потоком

2.DLL_THREAD_ATTACH – подключение потока. Когда происходит создание нового потока, который использует DLL. Это вызывается для каждого создаваемого потока. *Для главного потока не вызывается.*
3.DLL_THREAD_DETACH - при завершении потока с помощью функции ExitThread.
4.DLL_PROCESS_DETACH – при завершении процесса, если завершение потока включает завершение процесса. Если завершение процесса выполняется с помощью ExitProcess.
Вызов Exit… приводит к упорядоченному завершению.
Terminate… - просто вырубает поток, и могут не освободиться ресурсы (TerminateThread).
Следует избегать создания DllMain, т.к. она не является мобильным способом работы с DLL.

ЭКСПОРТ И ИМПОРТ ДАННЫХ

Вместо использования стандартных директив компилятора C++:
__declspec(dllexport)
__declspec(dllimport)
существует возможность альтернативного создания DLL с помощью ключевого слова extern. Его использование требует включение в проект DEF-файла. Этот способ в Win C++ считается устаревшим, т.к. не позволяет экспортировать/импортировать данные. При использовании нового способа данные экспортировать можно (как и функции):

```
__declspec(dllexport) int y;
__declspec(dllimport) int x;
```

Рекомендуется избегать экспорта/импорта данных (этот подход является непереносимым с платформы на платформу).

ЗАГРУЗКА DLL В ПАМЯТЬ

Загрузка Dll в Память, а также любых исполняемых модулей происходит методом отображения файла на адресное пространство процесса.

ПОЛЕЗНЫЕ ФУНКЦИИ ПРИ РАБОТЕ В DLL И ИСПОЛНЯЕМЫМИ МОДУЛЯМИ

```
HMODULE GetModuleHandle(
        LPCTSTR lpModuleName   // имя модуля
);
```

Проверяет, загружена ли библиотека в память и возвращает дескриптор этого модуля (если загружен).

```
DWORD GetModuleFileName(
  HMODULE hModule,    // контекст для модуля
  LPTSTR lpFilename,  // имя файла модуля
  DWORD nSize         // размер буфера
);
```

Возвращает полное имя загруженного модуля. Является незаменимой при анализе командной строки.

15. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом динамический импорта процедур.
1 и 2 часть вопроса смотри в вопросе номер 14

DLL представляет собой дополняемый модуль ОС (обычно DLL), код и ресурсы которого могут использоваться в составе других динамических библиотек и приложений. DLL – программа с множеством точек входа. В отличие от статической библиотеки, которая включается в выполняемый exe модуль на этапе сборки, динамическая библиотека собирается в виде отдельно выполняемого модуля. Использование динамической библиотеки выполняется одним из двух способов:

- Статического импорта
- Динамического импорта

Для создания DLL в разных языках используются разные средства. В C++:
Создание:

Динамический импорт. Если при статическом импорте загрузку DLL в память обеспечивает ОС, то при динамическом импорте это делает программист вручную. Загрузить DLL можно с помощью функции:
HANDLE LoadLibrary(LPCSTR libFileName)
Загрузка DLL-библиотеки в память:
HMODULE LoadLibrary(LPCTSTR lpFileName);
HMODULE LoadLibraryEx(LPCTSTR lpFileName, _Reserved_ HANDLE hFile, DWORD dwFlags);
HMODULE GetModuleHandle(LPCTSTR lpModuleName); GetModuleHandleEx.
DWORD GetModuleFileName(HMODULE hModule, LPTSTR lpFilename, DWORD nSize);
Освобождение DLL-библиотеки:
bool FreeLibrary(HMODULE hModule); FreeLibraryAndExitThread.
Получение адреса функции в DLL-библиотеке:
void* GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
Применение:

```
typedef int TMin(int x, int y); // добавить __stdcall
TMin* pMin;
pMin = (TMin*)GetProcAddress(hModule, "_Min@8");
int a = pMin(10, 20);
```

16. Понятие динамически-загружаемой библиотеки. Создание в DLL-библиотеке разделяемых между приложениями глобальных данных. Разделы импорта и экспорта DLL-библиотеки. Переадресация вызовов процедур DLL-библиотек к другим DLL-библиотекам. Исключение конфликта версий DLL.
DLL представляет собой дополняемый модуль ОС (обычно DLL), код и ресурсы которого могут использоваться в составе других динамических библиотек и приложений. DLL – программа с множеством точек входа. В отличие от статической библиотеки, которая включается в выполняемый exe модуль на этапе сборки, динамическая библиотека собирается в виде отдельно выполняемого модуля. Использование динамической библиотеки выполняется одним из двух способов:

- Статического импорта Динамического импорта

Запись о **переадресации** вызова функции (function forwarder) — это строка в разделе экспорта DLL, которая перенаправляет вызов к другой функции, находящейся в дру гой DLL. Например, запустив утилиту DumpBin из Visual C++ для Kernel32.dll в Win dows 2000, Здесь есть четыре переадресованные функции. Всякий раз, когда Ваше приложе ние вызывает *HeapAlloc*, *HeapFree*, *HeapReAlloc* или *HeapSize*, его EXE модуль динами чески связывается с Kernel32.dll. При запуске EXE модуля загрузчик загружает Ker nel32.dll и, обнаружив, что переадресуемые функции на самом деле находятся в NTDLL.dll, загружает и эту DLL. Обращаясь к *HeapAlloc*, программа фактически вызы вает функцию *RtlAllocateHeap* из NTDLL.dll. А функции *HeapAlloc* вообще нет!
При вызове *HeapAlloc* (см. ниже) функция *GetProcAddress* просмотрит раздел экс порта Kernel32.dll и, выяснив, что *HeapAlloc* — переадресуемая функция, рекурсивно вызовет сама себя для поиска *RtlAllocateHeap* в разделе экспорта NTDLL.dll. GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");
Вы тоже можете применять переадресацию вызовов функций в своих DLL. Самый простой способ — воспользоваться директивой *pragma*: // переадресация к функции из DllWork
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
Эта директива сообщает компоновщику, что DLL должна экспортировать функцию *SomeFunc*, которая на самом деле реализована как функция *SomeOtherFunc* в модуле DllWork.dll. Такая запись нужна для каждой переадресуемой функции.

Динамический импорт. Если при статическом импорте загрузку DLL в память обеспечивает ОС, то при динамическом импорте это делает программист вручную. Загрузить DLL можно с помощью функции:
HANDLE LoadLibrary(LPCSTR libFileName)
Имя файла отыскивается на диске в следующей последовательности:
1.Текущий каталог
2.Каталог системы Windows
3.Системный каталог Windows (system32)
4.В каталоге, содержащем исполняемый файл программы, вызвавшей функцию LoadLibrary
5.Во всех каталогах перечисленных в переменной среды PATH
6.В списке сетевых каталогов
После завершения работы с DLL вызывается void FreeLibrary(HANDLE);
Функция void* GetProcAddress(HANDLE,LPCSTR); По имени функции или ее номеру.
Ответственность за правильность вызова лежит на программисте.

- Разделяемые данные – shared data:
```
#pragma section("mysection", read, write, shared)
__declspec(allocate("mysection")) int Number = 0;
```

- Переадресация к процедуре в другой DLL:
1.#pragma comment(linker, "/export:MyProc=OtherDll.OtherProc")
2.В разделе экспорта DLL для процедуры MyProc создается переадресация к процедуре OtherProc в OtherDll.
3.Просмотр раздела экспорта:
C:\>dumpbin -exports MyDll.dll

- Исключение конфликта версий DLL:1. **c:\myapp\myapp.exe** загружает старую версию **c:\program files\common files\system\mydll.dll**, а должен загружать **mydll.dll** из своего же каталога.

2.Создать пустой файл **c:\myapp\myapp.exe.local**.
Будет грузиться библиотека **c:\myapp\mydll.dll**.
3.Создать каталог **c:\myapp\myapp.exe.local**.
Будет грузиться **c:\myapp\myapp.exe.local\mydll.dll**.
4.Создать файл манифеста для приложения. В этом случае .local файлы будут игнорироваться.

ФУНКЦИЯ ВХОДА/ВЫХОДА

DLL может иметь НЕОБЯЗАТЕЛЬНУЮ функцию BOOL WINAPI DllMain(HINSTANCE hInst, DWORD dwReason, LPVOID lpImpload), которая вызывается системой Windows автоматически в 4х случаях. hInst – дескриптор загружаемого модуля, который равен адресу, с которого DLL проецируется в память. dwReason – причина вызова функции (одна из 4х причин). lpImpload - показывает, как DLL загружается в память (методом неявной загрузки – статический импорт, или методом явной загрузки – динамический импорт). 4 причины вызова:
1.DLL_PROCESS_ATTACH – при первой загрузке DLL каким-либо потоком
2.DLL_THREAD_ATTACH – подключение потока. Когда происходит создание нового потока, который использует DLL. Это вызывается для каждого создаваемого потока. *Для главного потока не вызывается.*
3.DLL_THREAD_DETACH - при завершении потока с помощью функции ExitThread.
4.DLL_PROCESS_DETACH – при завершении процесса, если завершение потока включает завершение процесса. Если завершение процесса выполняется с помощью ExitProcess.
Вызов Exit… приводит к упорядоченному завершению.
Terminate… - просто вырубает поток, и могут не освободиться ресурсы (TerminateThread).
Следует избегать создания DllMain, т.к. она не является мобильным способом работы с DLL.

ЭКСПОРТ И ИМПОРТ ДАННЫХ

Вместо использования стандартных директив компилятора C++:
__declspec(dllexport)
__declspec(dllimport)
существует возможность альтернативного создания DLL с помощью ключевого слова extern. Его использование требует включение в проект DEF-файла. Этот способ в Win C++ считается устаревшим, т.к. не позволяет экспортировать/импортировать данные. При использовании нового способа данные экспортировать можно (как и функции):

```
__declspec(dllexport) int y;
__declspec(dllimport) int x;
```

Рекомендуется избегать экспорта/импорта данных (этот подход является непереносимым с платформы на платформу).

ЗАГРУЗКА DLL В ПАМЯТЬ

Загрузка Dll В Память, а также любых исполняемых модулей происходит методом отображения файла на адресное пространство процесса.
ПОЛЕЗНЫЕ ФУНКЦИИ ПРИ РАБОТЕ В DLL И ИСПОЛНЯЕМЫМИ МОДУЛЯМИ

```
HMODULE GetModuleHandle(
  LPCTSTR lpModuleName   // имя модуля
);
```

Проверяет, загружена ли библиотека в память и возвращает дескриптор этого модуля (если загружен).

```
DWORD GetModuleFileName(
  HMODULE hModule,    // контекст для модуля
  LPTSTR lpFilename,  // имя файла модуля
  DWORD nSize         // размер буфера
);
```

Возвращает полное имя загруженного модуля. Является незаменимой при анализе командной строки.

17. Понятие объекта ядра ОС Windows. Виды объектов ядра. Атрибуты защиты объекта ядра. Дескриптор защиты объекта ядра. Создание и удаление объектов ядра.

- Виды объектов:
  - Объекты оконной системы – User Objects
  - Объекты графической системы – GDI Objects
  - Объекты ядра – Kernel Objects

объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте.
Объекты ядра с атрибутами защиты:

- Атрибуты защиты – **SECURITY_ATTRIBUTES**:

```
struct SECURITY_ATTRIBUTES
{
   DWORD nLength;
   void* lpSecurityDescriptor;
   bool bInheritHandle;
};
```

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию, которая подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются.

- Объекты ядра можно защитить дескриптором защиты, который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений; создавая клиентское приложение, можно игнорировать это свойство объектов ядра.
- Дескриптор защиты – **SECURITY_DESCRIPTOR (структура не стандартизирована)**:
  - Owner security identifier (SID) – тот, кто запустил процесс
  - Primary group SID – основная группа, которой принадлежит пользователь
  - Discretionary access control list (DACL) – дать права на чтение/запись. В списке права на доступ. Этот список заполняет пользователь
  - System access control list (SACL) – этот - система
- Функции создания и редактирования дескриптора защиты:

bool **InitializeSecurityDescriptor**(PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD dwRevision);
SetSecurityDescriptorXxx, **GetSecurityDescriptorXxx**.

- Создание объекта ядра:
  Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, CreateFileMapping), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги.
  - HANDLE **CreateXxx**(SECURITY_ATTRIBUTES* lpAttributes, …, LPCTSTR lpName);
  - HANDLE **OpenXxx**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
  - bool **DuplicateHandle**(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, HANDLE hTargetProcessHandle, HANDLE* lpTargetHandle, DWORD dwDesiredAccess, bool bInheritHandle, DWORD dwOptions);
  - bool **SetHandleInformation**(HANDLE hObject, DWORD dwMask, DWORD dwFlags); HANDLE_FLAG_INHERIT, HANDLE_FLAG_PROTECT_FROM_CLOSE.
  - bool **GetHandleInformation**(HANDLE hObject, DWORD* lpdwFlags);
- Удаление объекта ядра:
  - bool **CloseHandle**(HANDLE hObject);

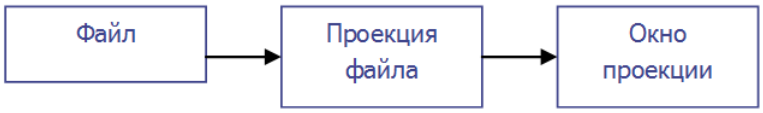
Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описатель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти. Если же описатель неверен, происходит одно из двух. В

нормальном режиме выполнения процесса CloseHandle возвращает FALSE, а GetLastError — код ERROR_INVALID_HANDLE. Но при выполнении процесса в режиме отладки система просто уведомляет отладчик об ошибке. Перед самым возвратом управления CloseHandle удаляет соответствующую запись из таблицы описателей: данный описатель теперь недействителен в Вашем процессе и использовать его нельзя. При этом запись удаляется независимо от того, разрушен объект ядра или нет! После вызова CloseHandle Вы больше не получите доступ к этому объекту ядра; но, если его счетчик не обнулен, объект остается в памяти. Тут все нормально, это означает лишь то, что объект используется другим процессом (или процессами). Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав CloseHandle), он будет разрушен. А вдруг Вы забыли вызвать CloseHandle — будет ли утечка памяти? И да, и нет. Утечка ресурсов (тех же объектов ядра) вполне вероятна, пока процесс еще исполняется. Однако по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежавшие этому процессу, и в случае объектов ядра действует так: в момент завершения процесса просматривает его таблицу описателей и закрывает любые открытые описатели.

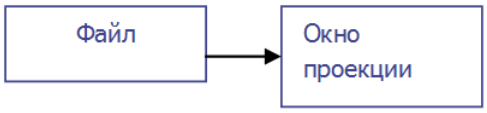
18.Проецирование файлов в память. Отличие в механизме проецирования файлов в память в ОС Windows и UNIX/Linux. Действия по проецированию файла в память.

На платформах Win/Unix существуют средства для работы с файлами как с оперативной памятью. Техника работы с ФПВП отличается в ОС Unix (Linux) & Win. В win сложнее: из неск. Пр-сов можно создать окно проекций одного файла => страницы проекции будут общие для всех, кто создал окно

В ОС Win отображение файлов в память является двухуровневым:



В UNIX схема проецирования файлов одноуровневая:



Идея в том, чтобы закрепить за началом файла какой-либо адрес памяти, а дальше выполнять чтение и запись файла методом чтения/записи байтов оперативной памяти. Т.к. файл не может поместится в оперативной памяти целиком, он делится на страницы и в оперативную память подгружаются лишь те страницы, с которыми происходит работа. Адресное пространство файла является виртуальным, оно может значительно превосходить по размерам оперативную память. Для прозрачной поддержки проецирования файлов в память необходимо иметь поддержку виртуальной памяти на уровне процессора и архитектуры компьютера. Оптимальное соотношение Вирт к физ памяти 1:1 – 1:1,25.

В ОС Win процессы работают в виртуальном адресном пространстве, для которого создается на диске файл подкачки (pagefile.sys). При проецировании файлов в память, файл подкачки не затрагивается, хотя проецирование происходит в виртуальное адресное пространство процесса. Такое возможно за счет аппаратной поддержки сложных таблиц страниц. В WIN – File Mapping; Unix – Memory Mapping.Для каждой страницы сохраняется ссылка на файл, из которого подгружается страница, поэтому файл подкачки не затрагивается. Файл подкачки – то, что спроецировала ОС в память. Если проецируем сами, он не используется

Хотя в Win существует OpenFile, но использовать ее не рекомендуется. Открытие/создание рекомендуется производить CreateFile. Для создания проекции важны первые 4 параметра.

2ой – указывается, будет файл читаться, записываться или и то и другое.

3ий – будет ли файл доступным для совместного использования со стороны других процессов. 0 – запретить сторонним процессам открывать этот файл.

4ый – атрибуты защиты.

Шаги: 1.От крытие файла CreateFile.

2.Создание объекта ядра под названием «проекция файла».

HANDLE CreateFileMapping(
HANDLE *hFile*, // Дескриптор файла полученный CreateFile
LPSECURITY_ATTRIBUTES *lpAttributes*, // Запись атрибутов защиты
DWORD *flProtect*, // Флаги.
DWORD *dwMaximumSizeHigh*, // Максимальный размер файла, для режимов, в которых возможна запись файла. Он может быть больше физического файла на диске. В этом случае размер дискового файла корректируется.
DWORD *dwMaximumSizeLow*,
LPCTSTR *lpName* // Имя объекта ядра.
);

Ос Win поддерживает файлы больше 4 Гб. Т.к. архитектура является 32х-разрядной, размеры файла задаются в виде 2х четырехбайтовых чисел. При работе с файлами > 4Гб существует проблема (на 32х разрядной платформе) – позиционирование в файле 32х-разрядное. Т.е. чтобы добраться до какого-то места в файле, нужно читать его последовательно.

3.Проецирование файла на физическую память:

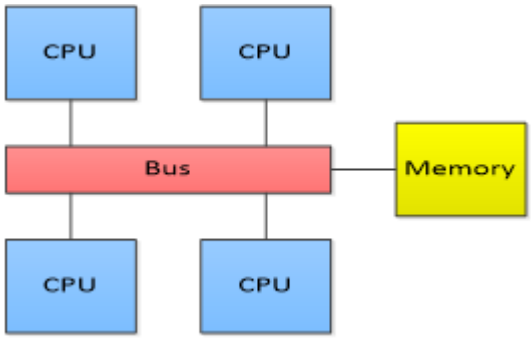
LPVOID MapViewOfFile(
HANDLE *hFileMappingObject*, // Дескриптор на проекцию файла.
DWORD *dwDesiredAccess*, // режим доступа.
DWORD *dwFileOffsetHigh*,

DWORD *dwFileOffsetLow*,
SIZE_T *dwNumberOfBytesToMap* // Размер окна проекции. – 32х-разрядное число.
);
Функция создает окно проекции в проекции физической памяти и возвращает его адрес.
BOOL UnmapViewOfFile(
LPCVOID *lpBaseAddress* //баз адрес
); //Закрывает окно проекции
BOOL FlushViewOfFile(
LPCVOID *lpBaseAddress*,
SIZE_T *dwNumberOfBytesToFlush* // размер в обл**); //Записывает Все изменения в файл**
Основное назначение файлов, проецируемых файлов – работа со сложными структурами данных, загрузка которых в память иным способом требует большого количества времени и сил.

4. Закрыть окно проекции UnmapViewOfFile(…)
5. Закрыть файл CloseHandle(HANDLE hObject)

Основное назначение файлов, проецируемых файлов – работа со сложными структурами данных, загрузка которых в память иным способом требует большого количества времени и сил. ОС исп-ет проецирование при исп-и DLL , они всегда загружаются при помощи проецирования!!!

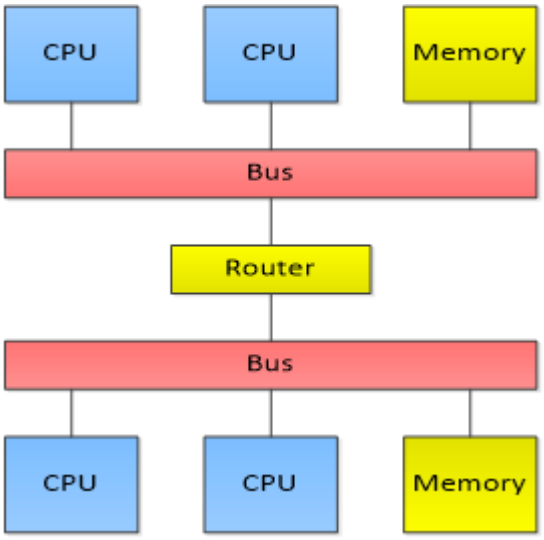
19. Современные многопроцессорные архитектуры SMP и NUMA. Многоуровневое кэширование памяти в современных процессорах. Проблема перестановки операций чтения и записи в архитектурах с ослабленной моделью памяти. Способы решения проблемы перестановки операций чтения и записи.



Производительность достигается за счет правильного проектирования, переноса ключевых функций в режим ядра. ОС Windows поддерживает много аппаратных архитектур. С целью наиболее эффективного использования аппаратных архитектур Windows построена как ОС с симметричной многопроцессорной обработкой (Symmetric MultiProcessing SMP)

В системах SMP на любом процессоре может работать поток ОС или прикладной программы. В системах с симметричной многопроцессорной обработкой потоки ОС работают на одном процессоре, а потоки прикладных программ на других процессорах. Если много процессоров, произв-ть => 0

NUMA (Non-Uniform Memory Architecture) Архитектура с неоднородной памятью



В компах с архитектурой NUMA существует возможность наращивать аппаратуру с помощью плат расширения. На каждой плате располагается блок из нескольких процессоров и блок памяти. В компах с такой архитектурой каждый процессор имеет доступ к любому блоку памяти на любой плате, однако доступ к памяти на своей плате выполняется быстрее. ОС Windows умеет так планировать потоки, что потоки, обращающиеся к общим участкам памяти, размещаются на процессорах одной платы, например, потоки одной прикладной программы целесообразно размещать на процессорах одной платы. **Суть** этой архитектуры – в особой организации памяти, а именно: *память физически распределена по различным частям системы, но логически она является общей,* так что пользователь видит единое адресное пространство.

Многоуровневое кэширование памяти в современных процессорах. Специфика конструирования современных ядер процессоров привела к тому, что систему кэширования в подавляющем большинстве процессоров приходится делать многоуровневой. Уровни кэш-памяти – L1, L2, L3. Кэш-память разных процессоров может быть когерентной и некогерентной;(свойство кэшей, означающее целостность данных, хранящихся в локальных кэшах для разделяемого ресурса). **Кэш первого уровня** в каждом ядре (самый «близкий» к ядру) разделяется на две (как правило, равные) части: *кэш команд (L1K) и кэш данных (L1D)(* «гарвардской структурой» ядер процессора). **кэш-память второго уровня** L2, как правило, в 8 раз больше по объёму, примерно втрое медленнее, и является уже «смешанной» — там располагаются и команды, и данные. Общей для всех ядер процессора является **кэш-память третьего уровня** L3, которая в 4-8 раз больше, чем кэш-память L2 (в расчете на одно ядро), и ещё втрое медленнее (но всё ещё быстрее оперативной памяти). Сначала информация ищется в кэш-памяти L1, L2, L3, потом в оперативной памяти.

Если кэш-память не является когерентной, в многопоточных приложениях может происходить **перестановка операций чтения и записи**. Также перестановки могут возникать благодаря оптимизациям компилятора. Рассмотрим пример – переменная модифицируется внутри блокировки (критической секции), затем блокировка снимается:

```
LOAD    [&value], %o0  // Загрузить значение переменной в регистр
ADD     %o0, 1, %o0 // Увеличить на единицу
STORE   %o0, [&value]  // Записать в переменную
STORE   0, [&lock]     // Отпустить блокировку
```

Важно, чтобы запись в переменную выполнилась до того, как выполнится запись, отпускающая блокировку. На архитектурах с ослабленной моделью памяти *(weak memory ordering)* другой процессор может увидеть отпущенную блокировку до того, как увидит новое значение переменной. Возможна ситуация, когда другой процессор захватит блокировку и увидит старое значение переменной.

Существуют следующие **решения данной проблемы**: Синхронизация кэшей – cache coherence; Барьеры памяти - memory barrier (memory fence) – операции до и после него переставить нельзя; Для барьера памяти выглядит так:

```
LOAD    [&value], %o0  // Загрузить значение переменной в регистр
ADD     %o0, 1, %o0 // Увеличить на единицу
STORE   %o0, [&value]  // Записать в переменную
MEMORYBARRIER          // Разделить операции барьером памяти
STORE   0, [&lock]     // Отпустить блокировку
```

В процессорах x86 и SPARC применяется **строгая модель памяти** (strong memory ordering), а именно, модель со строгим порядком записи – total store ordering (TSO):

- Чтения упорядочиваются в соответствии с предыдущими чтениями;
- Записи упорядочиваются в соответствии с предыдущими чтениями и записями;

Это означает, что чтения могут «проглядеть» предыдущие записи, но не могут проглядеть предыдущие чтения, а записи не могут «проглядеть» предыдущие чтения и записи. Если были только записи, то чтение могло прочитать не ту запись, но это не проблема, т.к. после каждой записи идёт её чтение из регистра соответствующим потоком. Если запись была сделана другим потоком. То ее можно «поглядеть» => надо синхр-ть потоки.

Если ОС ставит пр-с на др. ядро после кванта времени, то кэш обновляется

20. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Достоинства и недостатки процессов и потоков. Создание и завершение процесса. Запуск процессов по цепочке.

Многозада́чность — свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.
Существует 2 типа многозадачности:

- Процессная многозадачность (надёжно, но менее эфф-но)
- Поточная многозадачность (эфф-но, но менее надёжно)

Процесс – выполняемая программа, которая имеет свое виртуальное адресное пространство, выполняемый код, указатели на объекты ядра, данные, уникальный идентификатор, как минимум один выполняющий поток. Аварийное завершение процесса не приводит к утечке ресурсов или нарушению целостности данных в других процессах. Однако, поскольку процессы работают в разных адресных пространствах, необходимо использовать средства Inter-Process Communication (IPC) для доступа к общим данным.
Поток – выполняемая подпрограмма процесса, разделяющая с другими потоками общие ресурсы процесса. Все потоки процесса разделяют его вирт. Адресное пространство и системные ресурсы. Каждый поток имеет свой уникальный идентификатор. Расход памяти при распараллеливании минимален. Основные расходы памяти связаны с организацией стека на каждый параллельный поток. Производительность при работе с общими данными максимальна, поскольку потоки работают в общем адресном пространстве. Однако аварийное завершение потока часто приводит к утечке памяти процесса или даже к аварийному завершению процесса. Из-за общей памяти, целостность общих данных может быть нарушена.
Создание процессов
bool **CreateProcess**(LPCTSTR lpAppName, LPTSTR lpCmdLine,
SECURITY_ATTRIBUTES* lpProcessAttributes,
SECURITY_ATTRIBUTES* lpThreadAttributes,
bool bInheritHandles, DWORD dwCreationFlags,
void* lpEnvironment, LPCTSTR lpCurrentDirectory,
STARTUPINFO* lpStartupInfo,
PROCESS_INFORMATION* lpProcessInformation);
Процесс в *Windows* описывается структурой данных EPROCESS [5]. CreateProcess, CreateProcessAsUser, CreateProcessWithTokenW и CreateProcessWithLogonW. Создание процесса Windows состоит из нескольких этапов. создание нового процесса влечет за собой создание объектов ядра «процесс» и «поток». В момент создания система присваивает счетчику каждого объекта начальное значение — единицу. Далее функция CreateProcess (перед самым возвратом управления) открывает объекты «процесс» и «поток» и заносит их описатели, специфичные для данного процесса, в элементы hProcess и hThread структуры PROCESS_INFORMATION. Когда CreateProcess открывает эти объекты, счетчики каждого из них увеличиваются до 2.
Завершение процесса. Процесс можно завершить четырьмя способами:

1. входная функция первичного потока возвращает управление; единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку. При этом: любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами; система освобождает память, которую занимал стек потока; система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает входная функция; счетчик пользователей данного объекта ядра «процесс» уменьшается на 1.
2. один из потоков процесса вызывает функцию VOID ExitProcess(UINT fuExitCode); (нежелательный способ); Эта функция завершает процесс (после завершения всех его потомков) и заносит в параметр fuExitCode код завершения процесса.
3. поток др. пр-са вызывает TerminateProcess (тоже нежелательно); BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode); ее может вызвать любой поток и завершить любой процесс.
4. все потоки процесса умирают по своей воле (большая редкость). В такой ситуации (а она может возникнуть, если все потоки вызвали ExitThread или их закрыли вызовом TerminateThread) операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается коду завершения последнего потока.

Запуск процесса по цепочке:

CreateProcess(…, &pi); // PROCESS_INFORMATION pi;
CloseHandle(pi.hThread);// если не нужно, сразу закрыли, чтоб не расходовать реурсы
WaitForSingleObject(pi.hProcess);//пр-с свободен, когда все его потоки завершились. Ждём, пока он занят
GetExitCodeProcess(pi.hProcess, &exitCode); // DWORD exitCode; с каким кодом возвраьа заверш-ся пр-с
CloseHandle(pi.hProcess);..освоб-е ресурсов пр-са

Полезные функции:

```
HANDLE GetCurrentProcess(void); // CloseHandle ничего не делает
HANDLE GetCurrentThread(void);  // CloseHandle ничего не делает
DWORD GetCurrentProcessId(void);
DWORD GetCurrentThreadId(void);
void ExitProcess(UINT uExitCode); // код возврата всех потоков
bool TerminateProcess(HANDLE hProcess, UINT exitCode);
bool GetExitCodeProcess(HANDLE hProcess, DWORD* exitCode);
DWORD WaitForInputIdle(HANDLE hProcess, DWORD millisec);
```

21. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Создание и завершение потока. Приостановка и возобновление потока. Контекст потока.

Многозада́чность — свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.
Существует 2 типа многозадачности:

- Процессная многозадачность (надёжно, но менее эфф-но)
- Поточная многозадачность (эфф-но, но менее надёжно)

Процесс – выполняемая программа, которая имеет свое виртуальное адресное пространство, выполняемый код, указатели на объекты ядра, данные, уникальный идентификатор, как минимум один выполняющий поток. Аварийное завершение процесса не приводит к утечке ресурсов или нарушению целостности данных в других процессах. Однако, поскольку процессы работают в разных адресных пространствах, необходимо использовать средства Inter-Process Communication (IPC) для доступа к общим данным.
Поток – выполняемая подпрограмма процесса, разделяющая с другими потоками общие ресурсы процесса. Все потоки процесса разделяют его вирт. Адресное пространство и системные ресурсы. Каждый поток имеет свой уникальный идентификатор. Расход памяти при распараллеливании минимален. Основные расходы памяти связаны с организацией стека на каждый параллельный поток. Производительность при работе с общими данными максимальна, поскольку потоки работают в общем адресном пространстве. Однако аварийное завершение потока часто приводит к утечке памяти процесса или даже к аварийному завершению процесса. Из-за общей памяти, целостность общих данных может быть нарушена.
Создание потока. При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра «поток» — полный аналог объекта ядра «процесс».
HANDLE CreateThread(PSECURITY_ATTRIBUTES psa; DWORD cbStack; PTHREAD_START_ROUTINE pfnStartAddr;PVOID pvParam; DWORD fdwCreate; PDWORD pdwThreadID);
- Чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент bInheritHandle значением TRUE.
- Параметр ***cbStack*** определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек.
- Аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1 Мб).
- Параметр ***pfnStartAddr*** определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр ***pvParam*** идентичен параметру pvParam функции потока.
DWORD WINAPI **ThreadProc**(void* lpParameter);
- Параметр ***fdwCreate*** определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.
Жизненный цикл потока начинается тогда, когда программа создает новый поток.
Запрос перекочевывает исполняющей системе Windows, где диспетчер процесса выделяет пространство под объект потока и вызывает ядро для инициализации блока управления потока (KTHREAD)
Завершение потока. Поток можно завершить четырьмя способами:
1. функция потока возвращает управление (рекомендуемый способ); Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом: любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами; система корректно освобождает память, которую занимал стек потока; система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») — его и возвращает Ваша функция потока; счетчик пользователей данного объекта ядра «поток» уменьшается на 1.
2. поток самоуничтожается вызовом функции VOID *ExitThread(DWORD dwExitCode);* (нежелательный способ); При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++-ресурсы (например, объекты, созданные из C++-классов) не очищаются.

3. один из потоков данного или стороннего процесса вызывает функцию BOOL *TerminateThread(HANDLE hThread, DWORD dwExitCode);* (нежелательный способ); завершает поток, указанный в параметре hThread.
4. завершается процесс, содержащий данный поток (тоже нежелательно). Функции ExitProcess и TerminateProcess, тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежавших завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно — так, будто для каждого из них вызывается функция TerminateThread.
При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.
Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT (она определена в заголовочном файле WinNT.h). Эта структура содержится в объекте ядра «поток». Надо, чтоб потом корректно вернуть поток на процессор
Приостановка и возобновление потока
В объекте ядра «поток» имеется переменная — счетчик числа простоев данного потока. При вызове CreateProcess или CreateThread он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время: сразу после создания поток не готов к выполнению, ему нужно время для инициализации. Создав поток в приостановленном состоянии, Вы можете настроить некоторые его свойства (например, приоритет). Закончив настройку, Вы должны разрешить выполнение потока. Для этого вызовите ***DWORD ResumeThread(HANDLE hThread.*** Если вызов ResumeThread прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в ином случае — 0xFFFFFFFF. Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить вызовом ***DWORD SuspendThread(HANDLE hThread);***Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи — нет. Как и ResumeThread, функция SuspendThread возвращает предыдущее значение счетчика простоев данного потока.

22. Понятие пула потоков. Архитектура пула потоков. Операции с потоками при работе с пулом потоков.

Понятие. Иногда надо сделать какую-либо небольшую задачу асинхронной, но создание потока оказывается слишком накладным. Класс ThreadPool обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками. Если имеются небольшие задачи, которые требуют фоновой обработки, пул управляемых потоков — это самый простой способ воспользоваться преимуществами нескольких потоков.

Почему нужен пул потоков?

- Старт нового потока занимает много времени.
- Количество процессоров ограничено.

Архитектура пула потоков:

-Рабочие потоки вызывают callback-функции.

-Ожидающие потоки ждут на объектах ожидания.

- Очередь рабочих потоков (work queue).

-Стандартный пул потоков на каждый процесс.

-Менеджер рабочих потоков (worker factory).

-Стандартный размер пула потоков – 500 рабочих потоков.

-Pooled Threads: Improve Scalability With New Thread Pool APIs

-Thread Pooling

-Thread Pool API

Класс ThreadPool обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками. Если имеются небольшие задачи, которые требуют фоновой обработки, пул управляемых потоков — это самый простой способ воспользоваться преимуществами нескольких потоков. Например можно создавать объекты, выполняющие асинхронные задачи в потоках из пула потоков.

Когда не следует использовать потоки из пула потоков

Существует несколько случаев, в которых необходимо создание и управление собственным потоком вместо использования объекта потоков из потоков из пула.

- Необходимо наличие основного потока.
- Поток должен иметь определенный приоритет.
- Имеются задачи, которые приводят к блокировке потока на долгое время. Пул потоков имеет максимальное количество потоков, поэтому большое число заблокированных потоков в пуле потоков может не дать запуститься задачам.
- Необходимо поместить потоки в однопоточный апартамент. Все потоки ThreadPool находятся в многопоточном апартаменте.
- Необходимо иметь стабильную идентификацию, сопоставленную с потоком, или назначить поток задаче.

Характеристики пула потоков

-Потоки из пула потоков являются фоновыми потоками. Для каждого потока используется размер стека по умолчанию, поток запускается с приоритетом по умолчанию и находится в многопоточном апартаменте.

-Для каждого процесса существует только один пул потоков.

Исключения в потоках из пула потоков

Необработанные исключения в потоках из пула потоков приводят к завершению процесса. Существует три исключения из этого правила:

- Исключение ThreadAbortException создается в потоке пула потоков вследствие вызова перегрузки Abort.
- Исключение AppDomainUnloadedException создается в потоке пула потоков вследствие выгрузки домена приложения.
- Среда CLR или процесс основного приложения прерывает выполнение потока.

QueueUserWorkItem. принимает указатель на функцию с одним параметром. Указанная задача передается в пул потоков и будет выполнена в соответствии с указанными флагами (флаги помогают пулу потоков определить как лучше выполнить задачу).

Функция RegisterWaitForSingleObject позволяет указать задачу, которая будет выполняться по событию (Event, Mutex, Semaphore, Console input и прочее). Если событие не возникает, то задача выполняется по истечении указанного периода времени. Это, например, удобно использовать для асинхронного отображения видео кадров приходящих по сети. При получении кадра он выводится, а если кадров долго нет, то показывается специальный обновляемый кадр с сообщением о проблеме.

Ещё одна функция — CreateTimerQueueTimer — позволяет создать асинхронный таймер. В этом случае задача ставится в очередь на выполнение регулярно(если другое не задано) через указанный период времени. Уже ясно, что задача выполняется в отдельном потоке, в отличии от обычного таймера Windows.

23. Распределение процессорного времени между потоками ОС Windows. Механизм приоритетов. Класс приоритета процесса. Относительный уровень приоритета потока. Базовый и динамический приоритеты потока. Операции с приоритетами.

Распределение процессорного времени между потоками ОС Windows

ОС выделяет процессорное время всем активным потокам, исходя из их уровней приоритета (scheduling priority), которые изменяются от 0 (низший) до 31. Уровень 0 присваивается особому потоку, выполняющему обнуление неиспользуемых страниц памяти. Ни один другой поток не может иметь уровень приоритета 0. Для каждого уровня приоритета ОС ведет свою очередь потоков. При появлении потока с более высоким уровнем приоритета, текущий поток приостанавливается (не дожидаясь истечения кванта времени) и квант времени отдается приоритетному потоку. Пока в системе существуют потоки с более высоким приоритетом, потоки с более низкими приоритетами простаивают. Потоки с одинаковым приоритетом обрабатываются как равноправные.

Механизм приоритетов. Уровни приоритета := в 2 этапа

1. Пр-су := класс приоритета (приоритет пр-са)
2. Потоку := относительный уровень приоритета (приоритет внутри пр-са)
3. Результирующий приоритет = сумма этих 2 значений (на самом деле по таблице). На пересечении (1) и (2) сумма, но мб и абсолютное значение

НЕЛЬЗЯ поднимать приоритет, чтобы работало быстрее, т.к. др. важные потоки могут простаивать долгое время. С-ма может перестать работать. Надо понижать приоритеты мешающим потокам (временно) или повышать своему только временно!!!!

Классы приоритета:		**Относительные уровни приоритета:**	
IDLE_PRIORITY_CLASS	4	THREAD_PRIORITY_IDLE	1 // общий результат
BELOW_NORMAL_PRIORITY_CLASS		THREAD_PRIORITY_LOWEST	–2
NORMAL_PRIORITY_CLASS	8	THREAD_PRIORITY_BELOW_NORMAL	–1
ABOVE_NORMAL_PRIORITY_CLASS		THREAD_PRIORITY_NORMAL	+0
HIGH_PRIORITY_CLASS	13	THREAD_PRIORITY_ABOVE_NORMAL	+1
REALTIME_PRIORITY_CLASS	24	THREAD_PRIORITY_HIGHEST	+2
		THREAD_PRIORITY_TIME_CRITICAL	5 // общий результат

Динамический приоритет:

Когда окно потока активизируется или поток находится в состоянии ожидания сообщений и получает сообщение или поток заблокирован на объекте ожидания и объект освобождается, ОС увеличивает его приоритет на 2, спустя квант времени ОС понижает приоритет на 1, спустя еще квант времени понижает еще на 1.

Динамический приоритет потока не может быть меньше базового приоритета и не может быть больше приоритета с номером 15. ОС не выполняет корректировку приоритета для потоков с приоритетом от 16 до 31. Приоритеты с 16 по 31 – приоритеты реального времени, их использовать не рекомендуется, причем даже в тех случаях, когда программа выполняет критические по времени операции. Поток, выполняющийся с приоритетом реального времени будет иметь даже больший приоритет, чем драйвер мыши или клавиатуры и чем другие драйверы ОС.

Функции:

bool SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
DWORD GetPriorityClass(HANDLE hProcess);
bool SetThreadPriority(HANDLE hThread, int nPriority);
int GetThreadPriority(HANDLE hThread);
//динамический приоритет
bool SetProcessPriorityBoost(HANDLE hProcess, bool disablePriorityBoost); SetThreadPriorityBoost.
bool GetProcessPriorityBoost(HANDLE hProcess, bool* pDisablePriorityBoost); GetThreadPriorityBoost.
bool SwitchToThread(); // yield execution to another thread
void Sleep(DWORD dwMilliseconds);
Sleep(0) отдаст время, но потом доработает. Др. поток начинает новый квант времени, а не дорабатывает, как SwitchToThread
DWORD SleepEx(DWORD dwMilliseconds, bool bAlertable);

Sleep и Switch toThread исп-ся по очреди для проверки, освободилась ли перем-я spin-блокировки

24. Механизмы синхронизации потоков одного и разных процессов в ОС Windows. Обзор и сравнительная характеристика механизмов синхронизации.

Между потоками одного процесса:
- Критическая секция – Critical Section
- Ожидаемое условие – Condition Variable
- Атомарная операция – Interlocked (Atomic) Function
- Барьер синхронизации – Synchronization Barrier

Между потоками любых локальных процессов:
- Блокировка – Mutex
- Семафор – Semaphore
- Событие – Event
- Ожидаемый таймер – Waitable Timer

Между потоками удаленных процессов:
- Почтовый ящик – Mailslot
- Труба – Named/Unnamed Pipe
- Windows Socket

Критическая секция – небольшой участок кода, требующий монопольного доступа к каким-то общим данным.
Ожидаемое условие – механизм синхронизации, позволяющий потокам дождаться выполнения некоторого (сложного) условия. Состоит из критической секции и переменной условия.
Атомарная операция – простая операция над машинным словом, которая или выполняется целиком, или не выполняется вообще.
Ожидание. Объекты ядра Windows могут находится в одном из двух состояний:
- Свободном состоянии (signaled)
- Занятом (not signaled)

Синхронизация – ожидание освобождения объекта ядра
Блокировка – mutex (mutually exclusive), бинарный семафор. Используется для обеспечения монопольного доступа к некоторому ресурсу со стороны нескольких потоков (различных процессов).
Семафор – объект ядра, использующийся для учета ресурсов. Семафор имеет внутри счетчик. Этот счетчик снизу ограничен значением 0 (семафор занят) и некоторым верхним значением N. В диапазоне 1..N семафор является свободным. Семафоры можно считать обобщением блокировки на несколько ресурсов.
Событие – примитивный объект синхронизации, применяемый для уведомления одного или нескольких потоков об окончании какой-либо операции. Событие бывает двух типов:
- Событие со сбросом вручную – manual-reset event;
- Событие с автосбросом – auto-reset event.

Ожидаемый таймер – объект ядра, самостоятельно переходящий в свободное состояние в определенное время и/или через определенные промежутки времени.
Оконный таймер – механизм посылки таймерных сообщений через определенные промежутки времени.

25. Синхронизация потоков в пределах одного процесса ОС Windows. Критическая секция. Операции с критической секцией. Атомарные операции.

Критическая секция (блокировка) – небольшой участок кода, требующий монопольного доступа к каким-то общим данным.
Не секция в коде, а переменная. Не явл. частью API => изменять поля нельзя, но операции исп-ть можно.

```
struct CRITICAL_SECTION
{
   LONG LockCount;
   LONG RecursionCount;
   HANDLE OwningThread;
   HANDLE LockSemaphore;
   ULONG_PTR SpinCount;
};
void InitializeCriticalSection(CRITICAL_SECTION* lpCriticalSection);
void EnterCriticalSection(CRITICAL_SECTION* lpCriticalSection); //обычно LockCount – > 1 без обращения к ядру. Обращается к ядру только если надо ждать и создавать семафор
void LeaveCriticalSection(CRITICAL_SECTION* lpCriticalSection); //обычно LockCount – > 1 без обращения к ядру, поэтому работает быстро
bool TryEnterCriticalSection(CRITICAL_SECTION* lpCriticalSection); // попробовать. Если занята – делать другое. если занята при Enter, то снимается с пр-сорного времени
bool InitializeCriticalSectionAndSpinCount(CRITICAL_SECTION* lpCriticalSection, DWORD dwSpinCount);
SetCriticalSectionSpinCount.
```

При инициализ-и не устан-ся SpinCount!!!!можно отдельно установить SetCSSCили сразу при иниц-и, но без этого нельзя! ОС тавит поток в очередь при попытке доступа. Постановка тратит около 1000 кванов – теряется много времени. Но если перем-я свободна, то это пустые затраты. Поэтому при Enter ОС указывает счётчик, сколько подождать, чтоб поставить в очередь. Если за это время не освоб-сь, то в очередь, если освоб-сь, сразу заъватывает перем-ю.
Если в пр-соре 1 ядро, SpinCount не исп-ся, т.к. в этом нет смысла, ОС сама это опр-ет. Их можно иниц-ть, но исп-ся они не будут. ОС сама опр-ет кол-во квантов, чтоб ждать

Атомарная операция – простая операция над машинным словом, которая или выполняется целиком, или не выполняется вообще.

```
LONG InterlockedIncrement(LONG*Addend);
InterlockedDecrement, InterlockedAnd, InterlockedOr, InterlockedXor.
LONG InterlockedExchange(LONG* Target, LONG Value); InterlockedExchangePointer.// Value - > Target, Target - > возвращаем
LONG InterlockedCompareExchange(LONG* Destination, LONG Exchange, LONG Comparand);.
InterlockedCompareExchangePointer. // сравн. с Comparand, если ==, то кладёт и возвр-ет, если !=, только возвр-ет
InterlockedBitTestAnd(Set/Reset/Complement).
InterlockedXxx64, InterlockedXxxNoFence,
InterlockedXxxAcquire, InterlockedXxxRelease.
```

26. Синхронизация потоков в пределах одного процесса ОС Windows. Ожидаемое условие (монитор Хора). Операции с ожидаемым условием. Пример использования ожидаемого условия для синхронизации потоков.

Ожидаемое условие – механизм синхронизации, позволяющий потокам дождаться выполнения некоторого (сложного) условия. Состоит из критической секции и переменной условия. Нужен, если вход в кС условный. Исп-ся 2 перем-е (2 блок-ки), вычисл-е усл-ё происх. м/у ними. Вторая пер-я – CONDITION_VARIABLE.
void **InitializeConditionVariable**(CONDITION_VARIABLE* CondVariable); - инициализирует условную переменную
bool **SleepConditionVariableCS**(CONDITION_VARIABLE* CondVariable, CRITICAL_SECTION* CriticalSection, DWORD dwMilliseconds); - ожидания изменений условной переменной
bool **SleepConditionVariableSRW**(CONDITION_VARIABLE* CondVariable, SRWLOCK* SRWLock, DWORD dwMilliseconds, ULONG Flags); - когда ждём перем-ю усл-я, можно работать в паре с ReadWriteLock, а не с КС. Читать данные можно всем, чтение не блок-ся, пока нет записи
void **WakeConditionVariable**(CONDITION_VARIABLE* CondVariable);- после измен-я перем-й, освобождается критический раздел одного потока
void **WakeAllConditionVariable**(CONDITION_VARIABLE* CondVariable); - всех ожидающих потоков

```
// CRITICAL_SECTION criticalSection; CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try{
   while (DataDoesntSatisfyCondition()) // функция программиста
      SleepConditionVariableCS(&conditionVariable, &criticalSection, INFINITE); } // leave CS &
//waitForCondVar, EnterCS, здесь КС освоб-с на время, но при выходе из Sleep захватывается
catch (...){
   LeaveCriticalSection(&criticalSection);
   throw; }
LeaveCriticalSection(&criticalSection);
```

Др. поток заходит в КС

```
// CRITICAL_SECTION criticalSection; CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try{
   ChangeData(); // процедура программиста
   WakeAllConditionVariableCS(&conditionVariable);}
catch (...){
   LeaveCriticalSection(&criticalSection);
   throw;}
LeaveCriticalSection(&criticalSection);
```

На Sleep 1-й поток вышел из КС и заснул на CV, пока его не поменяют. Когда Wake, перем-я меняется и поток в Sleeo заходит снова в КС. Т.е. он сначала ждёт, но когда 2-й поток сделал Wake, он потом покидает КС. 1-й поток получает Wake, видит, что усл-е изменилось, становится в очередь и заходит в КС, когда 2-й поток из нее выйдет.
Т.о. в Sleep поток захватывает одну пер-ю, захватывает другую, ждёт, когда ее значение изменится, отпускает ее, захватывает снова первую и работает дальше. Это выполнено на уровне ОС, сразу встроено в ядро, происходит атомарно

Условные переменные по размеру равны указателям (точно так же, как и пуш-блокировки), избегают использования диспетчера, автоматически оптимизируют во время операций ожидания список ожиданий и защищают от сопровождений блокировки. Кроме того, условные переменные полностью используют события с ключом, а не обычный объект события, который бы использовался разработчиками по своему усмотрению, что еще больше оптимизирует код даже в случаях возникновения конкуренции.

27. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Перевод объекта ядра в свободное состояние. Объекты синхронизации: блокировки, семафоры, события.

В случае синхронизации потоков о каждом из объектов ядра говорят, что он находится либо в свободном (signaled state), либо в занятом состоянии (nonsignaled state). Так, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он навсегда остается в этом состоянии. Внутри этого объекта поддерживается булева переменная, которая при создании объекта инициализируется как FALSE («занято»). По окончании работы процесса ОС меняет значение этой переменной на TRUE, сообщая тем самым, что объект свободен.
Можно проверить значение булевой переменной, принадлежащей объекту ядра и сообщить системе, чтобы та перевела Ваш поток в состояние ожидания и автоматически пробудила его при изменении значения булевой переменной с FALSE на TRUE. Тогда появляется возможность заставить поток в родительском процессе, ожидающий завершения дочернего процесса, просто заснуть до освобождения объекта ядра, идентифицирующего дочерний процесс.
Синхронизация – ожидание освобождения объекта ядра:
DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds); – ждать освобождения объекта ядра hHandle, dwMillisec, - сколько времени (в миллисекундах) поток готов ждать
DWORD **WaitForMultipleObjects**(DWORD nCount,
const HANDLE* lpHandles, bool bWaitAll, DWORD dwMilliseconds);
WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_ABANDONED.
DWORD **WaitForSingleObjectEx**(HANDLE hHandle, DWORD dwMillisec, bool bAlertable);
WaitForMultipleObjectsEx.
СОБЫТИЯ - примитивный объект синхронизации, применяемый для уведомления одного или нескольких потоков об окончании какой-либо операции. Событие бывает двух типов: Событие со сбросом вручную – manual-reset event; Событие с автосбросом – auto-reset event.
- HANDLE **CreateEvent**(SECURITY_ATTRIBUTES* lpSecurityAttributes, bool bManualReset, bool bInitialState, LPCTSTR lpName); **OpenEvent**.
- bool **SetEvent**(HANDLE hEvent); bool **ResetEvent**(HANDLE hEvent);
- bool **PulseEvent**(HANDLE hEvent); – если это событие со сбросом вручную, то запускаются все ожидающие потоки; если это событие с автосбросом, то запускается лишь один из ожидающих потоков. Переводит событие в свободное, а потом снова в занятое
- bool **CloseHandle**(HANDLE hObject);
Пример. Прорисовать прогресс записи в БД. Пишущий поток шлёт периодически сообщения, чтоб другой прорисовал. Первый должен подождать отрисовки и только потом работать дальше, чтоб прогресс отображался правильно. Поэтому он создает Event, вкладывает в сообщ-е, ждёт его вып-я рисующим потоком (WaitForSO), а потом работает дальше. Второй поток, когда обработал сообщ-е с Event'ом, должен его освободить.
СЕМАФОР - объект ядра, использующийся для учета ресурсов. Семафор имеет внутри счетчик. Этот счетчик снизу ограничен значением 0 (семафор занят) и некоторым верхним значением N. В диапазоне 1..N семафор является свободным. Счётчик показывает кол-во свободных ресурсов. Если уничтожить поток, занимающий семафор, может произойти утечка сост-я семафора, т.к. остается всё меньше ресурсов в счётчике, хотя потоки, кот. их занимали, уже прибиты, но работать при этом нельзя.
- HANDLE **CreateSemaphore**(SECURITY_ATTRIBUTES* lpSecurityAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);
- HANDLE **OpenSemaphore**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
bool **ReleaseSemaphore**(HANDLE hSemaphore, LONG lReleaseCount, LONG* lpPreviousCount);
bool **CloseHandle**(HANDLE hObject);
МЬЮТЕКС - бинарный семафор, межпр-сная КС. Исп-ся для обесп-я монопольного доступа к нек. ресурсу со стороны нескольких потоков (различных процессов). Если ждём мьютекс, а занимающий его поток уничт-ся, то ожидающий поток получ. WAIT_ABANDONED, ОС сама освоб-ет мьютекс.
HANDLE **CreateMutex**(SECURITY_ATTRIBUTES* lpMutexAttributes,
bool bInitialOwner, LPCTSTR lpName);
HANDLE **OpenMutex**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
bool **ReleaseMutex**(HANDLE hMutex);
bool **CloseHandle**(HANDLE hObject);

28. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Ожидаемые таймеры. Оконные таймеры.

В случае синхронизации потоков о каждом из объектов ядра говорят, что он находится либо в свободном (signaled state), либо в занятом состоянии (nonsignaled state). Так, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он навсегда остается в этом состоянии. Внутри этого объекта поддерживается булева переменная, которая при создании объекта инициализируется как FALSE («занято»). По окончании работы процесса ОС меняет значение этой переменной на TRUE, сообщая тем самым, что объект свободен.
Можно проверить значение булевой переменной, принадлежащей объекту ядра и сообщить системе, чтобы та перевела Ваш поток в состояние ожидания и автоматически пробудила его при изменении значения булевой переменной с FALSE на TRUE. Тогда появляется возможность заставить поток в родительском процессе, ожидающий завершения дочернего процесса, просто заснуть до освобождения объекта ядра, идентифицирующего дочерний процесс.
Синхронизация – ожидание освобождения объекта ядра:
DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds); – ждать освобождения объекта ядра hHandle, dwMillisec, - сколько времени (в миллисекундах) поток готов ждать
DWORD **WaitForMultipleObjects**(DWORD nCount,
const HANDLE* lpHandles, bool bWaitAll, DWORD dwMilliseconds);
WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_ABANDONED.
DWORD **WaitForSingleObjectEx**(HANDLE hHandle, DWORD dwMillisec, bool bAlertable);
WaitForMultipleObjectsEx.
Ожидаемые таймеры (waitable timers) — это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени.
- HANDLE CreateWaitableTimer(PSECURITY_ATTRIBUTES psa, BOOL fManualReset, PCTSTR pszName);
- HANDLE OpenWaitableTimer(DWORD dwDesiredAccess, BOOL bInheritHandle, PCTSTR pszName);
По аналогии с событиями параметр fManualReset определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.
- Объекты «ожидаемый таймер» всегда создаются в занятом состоянии.
BOOL SetWaitableTimer(HANDLE hTimer, const LARGE_INTEGER *pDueTime, LONG lPeriod,
PTIMERAPCROUTINE pfnCompletionRoutine, PVOID pvArgToCompletionRoutine, BOOL fResume);
Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние
Эта функция принимает несколько параметров, в которых легко запутаться. Очевидно, что hTimer определяет нужный таймер. Следующие два параметра (pDueTime и lPeriod) используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

29. Структура системного программного интерфейса ОС Windows (Native API). Nt-функции и Zw-функции в пользовательском режиме и режиме ядра ОС Windows.

Все функции ядра Windows, доступные пользовательским приложениям, экспортируются библиотекой NtDll.dll. Системные функции называются **Native API**.
Native API - в основном недокументированный интерфейс программирования приложений (API), предназначенный для внутреннего использования в операционных системах семейства Windows NT, выпущенных Microsoft. В основном он используется во время загрузки системы, когда другие компоненты Windows недоступны, а также функциями системных библиотек (например, kernel32.dll), которые реализуют функциональность Windows API. Точкой входа программ, использующих Native API является функция DriverEntry(), так же как и в драйверах устройств Windows.
Как уже ранее упоминалось, пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как **Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll**.
Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:

- функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows;
- вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

Первая группа функций предоставляет интерфейс к службам исполняющей системы Windows, которые могут быть вызваны из пользовательского режима. К этой группе относятся более чем 400 функций, среди которых NtCreateFile, NtSetEvent и т. д. Как уже отмечалось, основная часть возможностей, присущих данным функциям, доступна через Windows API. Но некоторые возможности недоступны и предназначены для использования только внутри операционной системы.
Каждой Nt-функции сопоставлен номер. Номера зависят от версии Windows, полагаться на них не следует. Номер функции – это индекс в двух системных таблицах: nt!KiServiceTable и nt!KiArgumentTable. В первой таблице (System Service Descriptor Table – SSDT) хранятся адреса Nt-функций, во второй таблице – объемы параметров в байтах.
Для каждой из этих функций в Ntdll содержится точка входа с именем, совпадающим с именем функции. Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб, который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле Ntoskrnl.exe.
Исполняющая система Windows находится на верхнем уровне файла **Ntoskrnl.exe**. (Ядро составляет его нижний уровень.)
Системные функции имеют префикс Nt, например NtReadFile. В режиме ядра дополнительно существуют парные функции с префиксом Zw, например ZwReadFile. Они вызываются драйверами вместо Nt-функций. В пользовательском режиме Zw-имена тоже объявлены, но эквивалентны Nt-именам.
Если Nt-функция вызывается внутри ядра:

- Проверка параметров не выполняется.
- Такая функция может быть недоступна в ядре, т.е. может не экспортироваться модулем Ntoskrnl.exe.
- Вызов Nt-функции с передачей ей указателей на память ядра закончится ошибкой.

Вместо Nt-функций модули ядра вызывают Zw-функции. Zw-функция делает следующее:

- Загружает в регистр EAX номер функции.
- Загружает в регистр EDX указатель на вершину параметров в стеке ядра.
- Вызывает соответствующую ей Nt-функцию. При этом проверка параметров не выполняется.

30. Системный вызов ОС Windows. Алгоритм системного вызова. Особенность системного вызова из режима ядра.

Систе́мный вы́зов (англ. system call) - обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции (механизм, позволяющий пользовательским программам обращаться к услугам ядра ОС; интерфейс между операционной системой и пользовательской программой). С точки зрения программиста, системный вызов обычно выглядит, как вызов подпрограммы или функции из системной библиотеки. Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессорной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова.

Алгоритм системного вызова:

1. Загрузить в регистр EAX номер Nt-функции.

2. Загрузить в регистр EDX указатель на вершину параметров в стеке (ESP).

3. Вызвать прерывание для перехода процессора в режим ядра

4. Если используется прерывание (**int 0x2E** – на старых процессорах), то вызывается обработчик прерывания (Interrupt Service Routine – ISR), зарегистрированный в таблице обработчиков прерываний (Interrupt Descriptor Table – IDT) под номером 0x2E. Этот обработчик вызывает функцию ядра **KiSystemService**()

5. Если используется специальная инструкция (**sysenter** - на современных процессорах Intel или **syscall** – на современных процессорах AMD), то происходит вызов функции, адрес которой хранится в специальном внутреннем регистре процессора (Model Specific Register – MSR). Этот регистр хранит адрес функции ядра **KiFastCallEntry**()

6. После перехода в режим ядра все параметры, передаваемые в Nt-функцию, находятся на стеке ПР.

7. По номеру функции в регистре EAX отыскать в nt!KiArgumentTable количество байтов, занимаемое параметрами на стеке.

8. Скопировать параметры со стека ПР на стек ядра.

9. По номеру функции в регистре EAX отыскать в nt!KiServiceTable адрес функции для вызова.

10. Выполнить вызов функции.. Функция выполняется в контексте вызывающего процесса и потока и поэтому обращается к указателям пользовательского режима напрямую.

11. Если функция вызвана из ПР, выполняется проверка параметров. . Скалярные значения проверяются на допустимые диапазоны. Указатели проверяются с помощью функций ProbeForRead() и ProbeForWrite() в блоке __try { } __except { }.

12. Вернуться из режима ядра в ПР (**iret** – на старых процессорах, **sysexit** – на современных процессорах Intel, **sysret** – на современных процессорах AMD).Процессор восстанавливает из стека сохраненные значения регистров ПР и продолжает исполнение со следующей инструкции.

Системный вызов внутри ядра

Если Nt-функция вызывается внутри ядра: **1.** Проверка параметров не выполняется. **2.** Такая функция может быть недоступна в ядре, т.е. может не экспортироваться модулем Ntoskrnl.exe. **3.** Вызов Nt-функции с передачей ей указателей на память ядра закончится ошибкой.

Если вместо Nt-функций модули ядра вызывают Zw-функцию, она: **1.** Загружает в регистр EAX номер функции. **2.** Загружает в регистр EDX указатель на вершину параметров в стеке ядра. **3.** Вызывает соответствующую ей Nt-функцию. При этом проверка параметров не выполняется.

31. Отладка драйверов ОС Windows. Средства отладки драйверов. Посмертный анализ. Живая отладка.

Средства отладки драйверов:

Поддержка отладки ядра обеспечивается самим ядром Windows. Включается: bcdedit /debug on

В процессорах архитектуры x86 имеются специальные отладочные регистры DR0-DR7. Они позволяют отладчику ставить контрольные точки на чтение и запись памяти, а также на порты ввода-вывода.

Традиционный отладчик с пользовательским интерфейсом – windbg.exe.

Отладчик командной строки – kd.exe.

Современное средство отладки: Visual Studio 2013 Professional + WDK 8.1.

Начиная с Windows Vista, обеспечивается создание ряда драйверов, работающих в пользовательском режиме. Для отладки применяется Visual Studio.

Виды отладки в режиме ядра

Посмертный анализ (postmortem analysis):

Включить в операционной системе создание дампов памяти:
Start->Control Panel->System->Advanced system settings->Advanced tab
->Startup and Recovery->Settings->Write debugging information:
Small memory dump (256 KB) или Kernel memory dump.

Включить отладку в ядре: bcdedit /debug on

Настроить канал связи отладчика с ядром: bcdedit /dbgsettings

В отладчике включить загрузку символьной информации о ядре Windows: Tools->Options->Debugging->Symbols->[x] Microsoft Symbol Servers.

Открыть файл C:\Windows\MEMORY.DMP в отладчике.

Живая отладка (live debugging):

Соединить два компьютера через один из следующих интерфейсов:
Serial, IEEE 1394, USB 2.0.

Включить отладку в ядре: bcdedit /debug on

Настроить канал связи отладчика с ядром: bcdedit /dbgsettings

В отладчике включить загрузку символьной информации о ядре Windows: Tools->Options->Debugging->Symbols->[x] Microsoft Symbol Servers.

В Visual Studio собрать драйвер.

Поставить контрольную точку в исходном коде и установить драйвер.

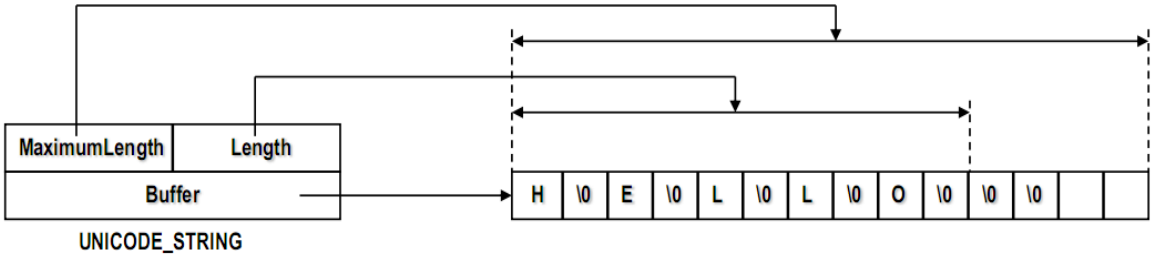
32.Структуры данных общего назначения в режиме ядра ОС Windows. Представление строк стандарта Unicode. Представление двусвязных списков.

UNICODE_STRING: Ядро Windows хранит строки в формате Unicode. Строки, передаваемые функциям ядра, находятся почти всегда в формате Unicode. Такие строки просты и логичны. Все символы в них представлены 16-битны- ми значениями (по 2 байта на каждый). В них нет особых байтов, указывающих, чем является следующий байт — частью того же символа или новым символом. Это значит, что прохождение по строке реализуется простым увеличением или уменьшением значения указателя. Так как каждый символ — 16-битное число, Unicode позволяет кодировать 65 536 символов

```
struct UNICODE_STRING
{    USHORT Length;
   USHORT MaximumLength;
   PWSTR Buffer;  };
```

Буфер, на который указывает поле Buffer, обычно выделяется из пула подкачиваемой страничной памяти.

Поле Length содержит число байтов (не WCHARS). Завершающий символ UNICODE_NULL в это число не включен.

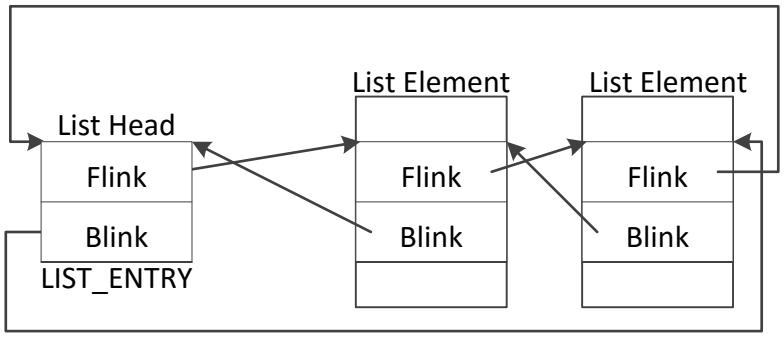


LIST_ENTRY: Большинство структур данных ядра хранятся как части двусвязных списков. Двусвязный список представляет собой список, связующая часть которого состоит из двух полей. Одно поле указывает на левого соседа данного элемента списка, другое — на правого. Кроме этого, со списком связаны два указателя — на голову и хвост списка
Преимущество использования двусвязных списков — в свободе передвижения по списку в обоих направлениях, в удобстве исключения элементов.

```
struct LIST_ENTRY
{
   LIST_ENTRY* Flink;
   LIST_ENTRY* Blink;
};
```

Поля Flink и Blink в структуре LIST_ENTRY указывают на вложенную структуру LIST_ENTRY, а не на начало элемента списка.

Структуры LIST_ENTRY никогда не содержат нулевых указателей. Когда список пуст, поля Flink и Blink в голове списка ListHead указывают непосредственно на ListHead.



33. Механизм прерываний ОС Windows. Аппаратные и программные прерывания. Понятие прерывания, исключения и системного вызова. Таблица векторов прерываний (IDT).

Прерывания представляют собой механизм позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния возникающие при работе процессора. Прерывания– это принудительная передача управления от выполняющейся программы к системе, а через неё к соответствующей программе обработки прерываний,происходящая при определенном событии. Основная цель введения прерываний –реализация асинхронного режима работы и распараллеливания работы отдельных устройств вычислительного комплекса. Механизм прерываний реализуется аппаратно-программными средствами. В зависимости от источника возникновения сигнала прерывания делятся на:

- асинхронные или внешние (аппаратные) — события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши;
- внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение, обращение к недопустимым адресам или недопустимый код операции;
- программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания как правило используются для обращения к функциям встроенного программного обеспечения

Структуры систем прерываний могут быть самыми различными, но все они имеют общую особенность – прерывание непременно ведет за собой изменение порядка выполнения команд процессором. Механизм обработки прерываний включает в себя следующие элементы

1. Установление факта прерывания (прием и идентификация сигнала на прерывание).
2. Запоминание состояния прерванного процесса (состояние процесса определяется значением счетчика команд, содержимым регистра процессора, спецификацией режима: пользовательский или привилегированный)
3. Управление аппаратно передается программе обработки прерывания. В этом случае, в счетчик команд заносится начальный адрес подпрограммы обработки прерывания, а в соответствующие регистры их слова состояния.
4. Сохранение информации прерванной программе, которую не удалось спасти с помощью действий аппаратуры.
5. Обработка прерывания. Работа может быть выполнена той же подпрограммой, которой было передано управление на 3-ем шаге, но в ОС чаще всего эта обработка реализуется путем вызова соотв. подпрограммы.
6. восстановление информации относящейся к прерванному процессу.
7. Возврат в прерванную программу.

Первые 3 шага реализуются аппаратными средствами, а остальные – программно. Главные функции механизма прерывания:

1. Распознавание или классификация прерывания.
2. Передача управления обработчику прерывания.
3. Корректное возвращение к прерванной программе Переход от прерванной программе к обработчику и обратно должен производится как можно быстрее. Одним из быстрых методов является использование таблицы сод.перечень всех допустимых для компьютера прерываний и адреса соотв.обработчиков. Для корректного возвращения к прерванной программе, перед передачей управления обработчику, содержимое регистров процессора запоминается либо в памяти с прямым доступом либо в системном стеке.

Прерывание – внешнее или внутреннее асинхронное событие;

Исключение – внешнее или внутреннее синхронное событие;

Системный вызов – внутреннее синхронное событие.

Исключительная ситуация (exception) - событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца.

Системные вызовы (system calls) - механизм, позволяющий пользовательским программам обращаться к услугам ядра ОС, то есть это интерфейс между операционной системой и пользовательской программой.

34.Аппаратные прерывания. Программируемый контроллер прерываний. Механизм вызова прерываний. Обработка аппаратных прерываний. Понятие приоритета прерываний (IRQL). Понятие процедуры обработки прерываний (ISR).

Таблица прерываний - Для того чтобы связать адрес обработчика прерывания с номером прерывания, используется таблица векторов прерываний, занимающая первый килобайт оперативной памяти - адреса от 0000:0000 до 0000:03FF. Таблица состоит из 256 элементов - FAR-адресов обработчиков прерываний. Вообще в таблице хран-ся ук-ли на объекты прер-й, каждый из кот. хранит данные и код. Именно код зарег-н в IDT. Эти элементы называются векторами прерываний. В первом слове элемента таблицы записано смещение, а во втором - адрес сегмента обработчика прерывания.Прерыванию с номером 0 соответствует адрес 0000:0000, прерыванию с номером 1 - 0000:0004 и т.д. Для программиста, использующего язык Си, таблицу можно описать следующим образом: void (* interrupt_table[256])(); Инициализация таблицы происходит частично BIOS после тестирования аппаратуры и перед началом загрузки операционной системой, частично при загрузке DOS. DOS может переключить на себя некоторые прерывания BIOS.

Обработка аппаратных прерываний:

Внешние прерывания поступают по своим линиям на программируемый контроллер прерываний – **Programmable Interrupt Controller (PIC).** В современных компьютерах используется Advanced PIC (APIC). Контроллер прерываний в свою очередь выставляет запрос на прерывание (Interrupt Request – IRQ) и посылает сигнал процессору по единственной линии. Процессор прерывает выполнение текущего потока, переключается в режим ядра, выбирает из контроллера запрос IRQ, транслирует его в номер прерывания, использует этот номер как индекс в таблице обработчиков прерываний, выбирает из таблицы адрес обработчика и передает на него управление. ОС программирует трансляцию номера IRQ в номер прерывания в IDT и устанавливает для прерываний приоритеты – **Interrupt Request Levels (IRQLs).**

-- 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 контроллером до завершения обработки всех более приоритетных прерываний. Затем, если менее приоритетные прерывания происходили, они материализуются контроллером.

-- Когда процессор обслуживает прерывание, считается, что он находится на уровне приоритета прерывания. На много-процессорных системах каждый процессор может находиться на своем IRQL. Для процессоров x86 установлено 32 приоритета, для процессоров x64 – 15 приоритетов (см. таблицу IRQL). Низший приоритет 0 (PASSIVE_LEVEL) обозначает работу вне обработчика прерываний.

Код режима ядра может менять IRQL процессора с помощью функций **KeRaiseIrql**() и **KeLowerIrql**(), расположенных в HAL. Обычно это происходит неявно при вызове обработчиков прерываний. Обработчик прерывания поднимает уровень прерывания перед началом работы и опускает уровень в конце работы.

Прерывание *(Interrupt) – сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся процедуре обработки прерывания, соответствующая данному событию, после чего исполнение кода продолжается ровно с того места где он был прерван (возвращение управления)*. **Процедура обработки прерывания** (Interrupt Service Routine) – это ни что иное как функция/подпрограмма, которую следует выполнить при возникновении определенного события. Будем использовать именно слово “процедура”, для того чтобы подчеркнуть ее отличие от всех остальных функций.

Главное отличие процедуры от простых функций состоит в том что вместо обычного “возврата из функции” (ассемблерная команда RET), следует использовать “возврат из прерывания” (ассемблерная команда RETI) – “REТurn from Interrupt**“.Прототип процедуры обработки прерывания**

Чтобы объявить некоторую функцию в качестве процедуры обработки того или иного прерывания, необходимо следовать определенным правилам прототипирования, чтобы компилятор/компоновщик смогли правильно определить и связать нужное вам прерывание с процедурой ее обработки. Во первых процедура обработки прерывания не может ничего принимать в качестве аргумента (void), а также не может ничего возвращать (void). Это связано с тем что все прерывания в AVR асинхронные, поэтому не известно в каком месте будет прервано исполнение программы, у кого принимать и кому возвращать значение, а также для минимизации времени входа и выхода из прерывания. **void** isr(**void**) Во вторых, перед прототипом функции следует указать что она является процедурой обработки прерывания. Как вам известно, в языке Си исполняется только тот код что используется в функции **main**. Поскольку процедура обработки прерывания в функции main нигде не используется, то для того чтобы компилятор не “выкинул” ее за ненадобностью, перед прототипом процедуры следует указать что эта функция является процедурой обработки прерывания.

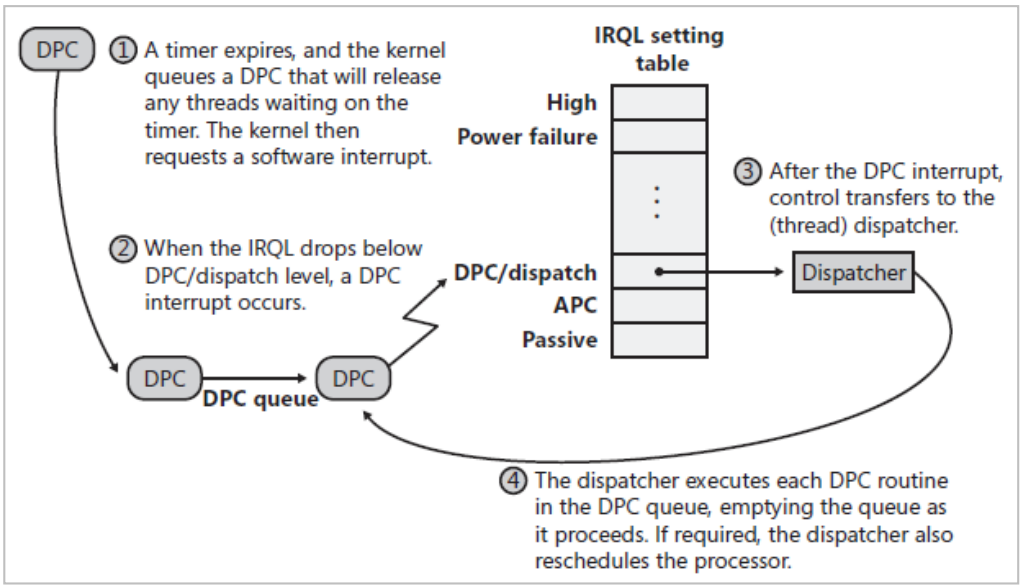
35. Понятие приоритета прерываний (IRQL). Приоритеты прерываний для процессора x86 или x64. Процедура обработки прерываний (ISR). Схема обработки аппаратных прерываний.

Приоритеты прерываний – Interrupt Request Levels (IRQLs):
-- 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 контроллером до завершения обработки всех более приоритетных прерываний. Затем, если менее приоритетные прерывания происходили, они материализуются контроллером. Это происходит от более приоритетных прерываний к менее приоритетным.
-- Когда процессор обслуживает прерывание, считается, что он находится на уровне приоритета прерывания. На много-процессорных системах каждый процессор может находиться на своем IRQL. Для процессоров x86 установлено 32 приоритета, для процессоров x64 –15 приоритетов. Низший приоритет 0 (PASSIVE_LEVEL) обозначает работу вне обработчика прерываний.
-- Код режима ядра может менять IRQL процессора с помощью функций **KeRaiseIrql**() и **KeLowerIrql**(), расположенных в HAL. Обычно это происходит неявно при вызове обработчиков прерываний. Обработчик прерывания поднимает уровень прерывания перед началом работы и опускает уровень в конце работы.
<u>На x86</u>: Уровни аппаратных прерываний
31 – HIGH_LEVEL: немаскируемое прер-е: всё плохо, с-ма остан-ся (сбой памяти и т.д.) Выз-ся самой с-мой, аппаратно
30 – POWER_LEVEL: пропадание электропитания, никогда не исп-сь. Это если бы не было бесперебойного питания, выдернуть вилку из розетки
29 – IPI_LEVEL: один пр-сор дёргает другой, чтоб обновить что-то или ещё что-то, например, обновить кэш обращ-й к данным
28 – CLOCK_LEVEL: ч-з него ОС выставляет потоки на пр-сор. Потоки выставл-ся только по прер-ям. По сути, самое приоритетное прер-е, если работать на одном пр-соре, то точно
27 – PROFILE_LEVEL: аппаратное профилирование ядра. Собир-ся инф-я о ф-ях и данных в стеке. Периодически дёргается, чтоб узнать, чем занимается ядро, кто работает в стеке
3-26 – DEVICE_LEVEL: назнач-ся ОС-мой. На x86: 1 пр-сор – уровень = 27 – IRQ number, неск. Пр-соров – round-robin в жиапазоне IRQL устр-в (в диапазоне 3 -26) (устр-во достали, вставили – уровень может измен-ся. Если что-то изменилось в BIOS, тоже. На x64/IA64 – IRQ number/16
Программные прерывания – не значит, что ч-з int дёгаем прер-я
2 – DISPATCH(dpc)_LEVEL: работает планировщик потоков (диспетчер) и отложенные пр- ры в ядре
1 – APC_LEVEL: уровень асинхр-х пр-р. У потока есть очередь асинхр. Пр-р режима ядра, туда ставят пр-ры драйверы. Эти пр-ры вып-ся с приритетом 1, более высоким, чем обычные пр-ры. Они прер-ют весь остальной код
0 – PASSIVE_LEVEL(low_level)
НЕ ПУТАТЬ ПРИОРИТЕТЫ ПРЕР-Й И ПРИОРИТЕТЫ ПОТОКОВ (ОНИ НА 0 УРОВНЕ)!!!
<u>Процедура обработки прерывания</u> (Interrupt Service Routine) – это ни что иное как функция/подпрограмма, которую следует выполнить при возникновении определенного события. Будем использовать именно слово “процедура”, для того чтобы подчеркнуть ее отличие от всех остальных функций. **Главное отличие процедуры от простых функций состоит в том что вместо обычного “возврата из функции” (ассемблерная команда RET), следует использовать “возврат из прерывания” (ассемблерная команда RETI) – “**RETurn from Interrupt**“.**
<u>Особенности процедур обработки прерываний:</u>
-- на уровнях аппаратных прерываний, а также на уровне программного прерывания DISPATCH_LEVEL, нельзя выполнять ожидания объектов, требующие переключения процессора на другой поток.
--Переключение на другой поток выполняется планировщиком на уровне прерываний DISPATCH_LEVEL, который в данном случае оказывается замаскирован, и поэтому возникает блокировка.
--в процедурах обработки аппаратных прерываний (и на уровне DISPATCH_LEVEL) можно работать лишь с физической памятью (non-paged memory), т.к. что попытка доступа к странице, которой нет в памяти, вызывает прерывание, в ответ на которое менеджер памяти вынужден инициировать подкачку страницы с диска и подождать завершения операции. Ожидание означает переключение на другой поток через вызов программного прерывания уровня DISPATCH_LEVEL, которое оказывается замаскированным.
--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. Она определяет допустимый уровень IRQLs для каждой API-функции ядра.

Обработка аппаратных прерываний:
Внешние прерывания поступают по своим линиям на программируемый контроллер прерываний – **Programmable Interrupt Controller (PIC).** В современных компьютерах используется Advanced PIC (APIC). Контроллер прерываний в свою очередь выставляет запрос на прерывание (Interrupt Request – IRQ) и посылает сигнал процессору по единственной линии. Процессор прерывает выполнение текущего потока, переключается в режим ядра, выбирает из контроллера запрос IRQ, транслирует его в номер прерывания, использует этот номер как индекс в таблице обработчиков прерываний, выбирает из таблицы адрес обработчика и передает на него управление. ОС программирует трансляцию номера IRQ в номер прерывания в IDT и устанавливает для прерываний приоритеты – Interrupt Request Levels (IRQLs).

36. Программные прерывания. Понятие отложенной процедуры (DPC). Назначение отложенных процедур. Механизм обслуживания отложенных процедур. Операции с отложенными процедурами.

Процедуры обработки прерываний (ISRs), работающие с высоким уровнем приоритета (IRQL), могут блокировать выполнение других процедур обработки прерываний с более низким приоритетом. Это увеличивает долю системных операций в общем времени работы приложений, т.е. увеличивает задержку (latency) в системе. Для разрешения такого типа проблем, программный код, предназначенный для работы в режиме ядра, должен быть сконструирован таким образом, чтобы избегать продолжительной работы при повышенных уровнях IRQL. Одним из самых важных компонентов этой стратегии являются Deferred Procedure Calls (DPC) — **отложенные процедурные вызовы**. Они представляется структурой ядра KDPC, в которой хранится адрес callback-процедуры, составляющей подпрограмму DPC.
Схема применения отложенных процедурных вызовов позволяет построить процесс выполнения таким образом, что задача может быть запланирована кодом, работающим на высоком уровне IRQL, но при этом еще не выполняется. Такая отсрочка выполнения применима, когда обработка данной ситуации может быть безболезненно перенесена на более позднее время. В таком случае процедура обработки прерываний выполняет абсолютный минимум работ с высоким уровнем приоритета. Для учета заявок на вызов DPC процедур операционная система поддерживает очередь объектов DPC (т.е. ОС ставит в очередь к процессору отложенную процедуру для выполнения работы на уровне приоритета DPC_LEVEL / DIPATCH_LEVEL; при постановке DPC в очередь, указывается один из трех приоритетов: Low, Medium – в конец очереди, High – в начало очереди), что позволяет системе быстро обработать другие прерывания. На время работы подключает к процессору отдельный стек DPC-процедур.
Объект DPC для использования в процедурах обработки прерываний создается по вызову IoInitializeDpcRequest, выполняемому обычно в стартовых процедурах драйвера. Данный вызов регистрирует предлагаемую драйвером DpcForIsr процедуру и ассоциирует ее с создаваемым объектом (DPC объект, созданный данным вызовом, останется недоступным разработчику драйвера; отличие DpcForIsr от других DPC-процедур состоит только в том, что работа с последними проходит при помощи вызовов Ke...Dpc , т.е. при вызове KeInitializeDpc(), KeInsertQueueDpc(), KeRemoveQueueDpc(), KeSetTargetProcessorDpc(), KeSetImportanceDpc(), KeFlushQueuedDpcs(); создаваемые для них DPC объекты доступны разработчику драйвера). Если драйвер зарегистрировал свою процедуру DpcForIsr, то во время обработки прерывания ISR процедурой в системную очередь DPC может быть помещен соответствующий DPC объект (запрос на вызов этой DpcForIsr процедуры позже) при помощи вызова IoRequestDpc. Процедура DpcForIsr завершит позже обработку полученного ISR процедурой запроса, что будет выполнено в менее критичных условиях и при низком уровне IRQL.
Функционирование DPC процедур: 1. Когда некоторый фрагмент программного кода, работающий на высоком (аппаратном) уровне IRQL желает запланировать выполнение части своей работы так, чтобы она была выполнена при низком значении IRQL, то он добавляет DPC объект в системную очередь отложенных процедурных вызовов. **2.** Когда значение IRQL процессора падает ниже DISPATCH_LEVEL, работа, которая была отложена прерыванием, обслуживается DPC функцией. Диспетчер DPC извлекает каждый DPC объект из очереди и вызывает соответствующую функцию, указатель на которую хранится в этом объекте (вызов выполняется в то время, когда процессор работает на уровне DISPATCH_LEVEL). **3.** Отложенная процедура выполняется на уровне прерываний DPC_LEVEL / DIPATCH_LEVEL на том же процессоре, что и вызывающая ISR, или на заданном процессоре.



DPC-пр-ра не может захватывать объекты ядра, выполнять ожидания объектов, обращаться к отсутствующим страницам памяти, делать системные вызовы.

37. Понятие асинхронной процедуры (APC). Назначение асинхронных процедур. Типы асинхронных процедур. Операции с асинхронными процедурами.

1)Как и DPC, применяется для выполнения отложенных действий.
2)Выполняется на уровне прерываний APC_LEVEL или PASSIVE_LEVEL в контексте заданного потока, и соотв., в виртуальном адресном пространстве процесса, которому принадлежит поток.
3)не подвержена ограничениям DPC-процедур, может захватывать объекты ядра, выполнять ожидания объектов, обращаться к отсутствующим страницам памяти, делать системные вызовы.

APC-процедура представляется стр-рой ядра KAPC, кот. содержит указатели на 3 подпрограммы:
1) RundownRoutine – выполняется, если из-за удаления потока удаляется структура KAPC.
2) KernelRoutine – выполняется на уровне приоритета APC_LEVEL.
3) NormalRoutine – выполняется на уровне приоритета PASSIVE_LEVEL.
Стр-ра KAPC создается и ставится в одну из двух очередей потока: одна очередь предназначена для APC режима ядра, вторая – для APC пользовательского режима. Начала очередей находятся в массиве из двух элементов: KTHREAD.ApcState.ApcListHead[].

Типы APC-процедур:
APC режима ядра – Kernel Mode APC, подразделяется на:

- Специальную APC – Special Kernel Mode APC
- Нормальную APC – Normal Kernel Mode APC

APC пользовательского режима – User Mode APC.

Если в очередь потоку ставится APC, и поток находится в состоянии ожидания объекта ядра, APC-процедура все-таки не заставит поток проснуться для ее выполнения. Чтобы поток просыпался для выполнения APC-процедур, он должен ожидать объекты с помощью alertable-функций, например WaitForSingleObjectEx(…, true), SleepEx(…, true).

Если у потока в очереди есть APC-процедура, и происходит переключение процессора на поток, APC-процедура получает приоритет над остальным кодом потока.

Функции управления APC режима ядра:

KeInitializeApc()

KeInsertQueueApc()

KeRemoveQueueApc()

KeFlushQueueApc()

KeAreApcsDisabled()

KeEnterGuardedRegion()

KeLeaveGuardedRegion()

KeEnterCriticalRegion()

KeLeaveCriticalRegion()

В пользовательском режиме:

QueueUserApc()

38. Понятие асинхронной процедуры (APC). Асинхронные процедуры режима ядра: специальная и нормальная APC-процедуры. Асинхронные процедуры пользовательского режима.

1)Как и DPC, применяется для выполнения отложенных действий.
2)Выполняется на уровне прерываний APC_LEVEL или PASSIVE_LEVEL в контексте заданного потока, и соотв., в виртуальном адресном пространстве процесса, которому принадлежит поток.
3)не подвержена ограничениям DPC-процедур, может захватывать объекты ядра, выполнять ожидания объектов, обращаться к отсутствующим страницам памяти, делать системные вызовы.

Специальная APC-процедура режима ядра:
- KAPC.KernelRoutine выполняется на уровне приоритета APC_LEVEL.
- KAPC.NormalRoutine == NULL.
- Помещается в очередь APC режима ядра после других специальных APC.
- Вызывается перед нормальными APC режима ядра.
- Вызывается, если IRQL == PASSIVE_LEVEL и поток не находится в защищенной секции (guarded region) – KTHREAD.SpecialApcDisable != 0.
- APC не может захватить блокировку, работающую на IRQL == 0.
- используется для завершения процесса, для передачи результатов ввода-вывода в адресное пространство потока.

Нормальная APC-процедура режима ядра:
- KAPC.NormalRoutine выполняется на уровне приоритета PASSIVE_LEVEL.
- Вызывается, если IRQL == PASSIVE_LEVEL, поток не находится в защищенной или критической секции (critical region) – KTHREAD.KernelApcDisable != 0, и не выполняет специальную APC ядра.
- Нормальной APC разрешено делать все системные вызовы.
- Норм.APC исп-тся ОС для завершения обработки запроса от драйвера – Interrupt Request Packet.

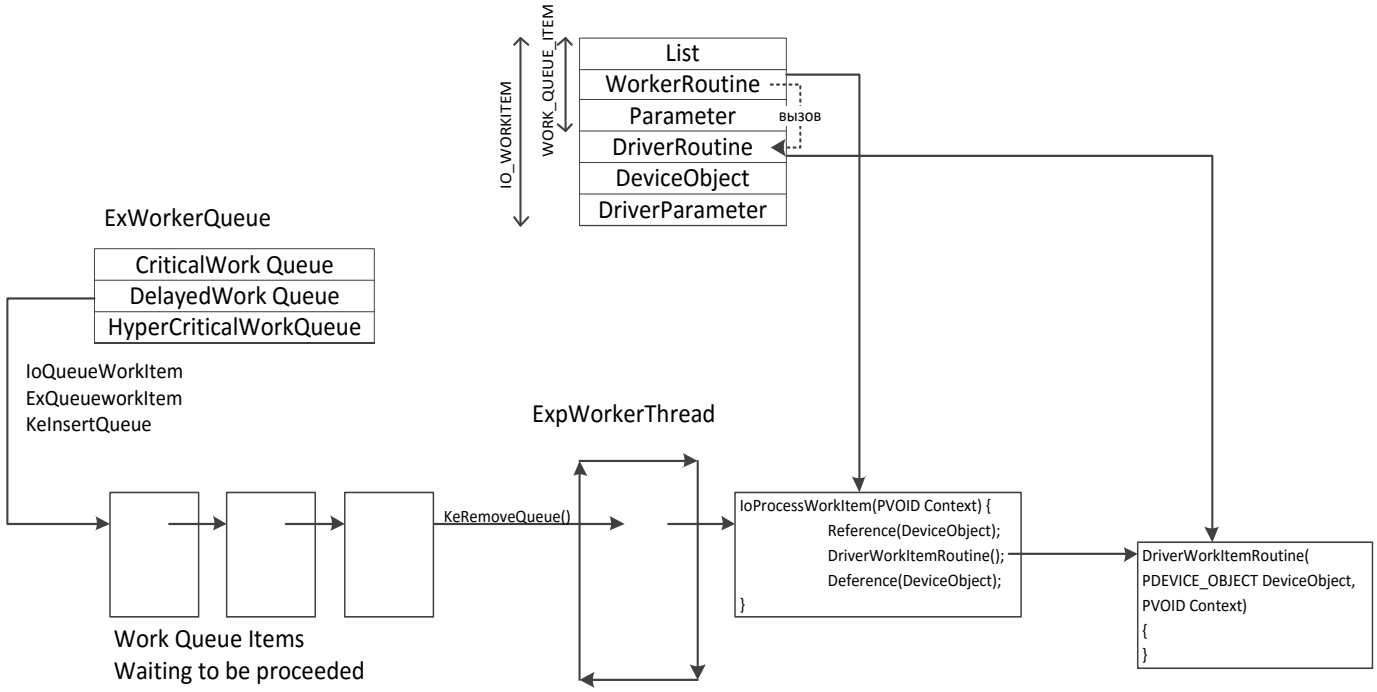
APC-процедура пользовательского режима:
- KAPC.NormalRoutine выполняется на уровне приоритета PASSIVE_LEVEL.
- Вызывается, только если поток ожидает объект ядра в alertable-режиме, например WaitForSingleObjectEx(…, true), WaitForMultipleObjectsEx(…, true), SleepEx(…, true).
- Создается вызовом процедуры QueueUserApc().
- Используется ОС, чтобы выполнить в пользовательском режиме подпрограмму завершения асинхронного ввода-вывода – I/O Completion Routine.
- Поле KAPC.KernelRoutine может содержать адрес дополнительной подпрограммы, выполняемой на уровне приоритета IRQL == APC_LEVEL перед выполнением подпрограммы KAPC.NormalRoutine. Эта подпрограмма выполняется только тогда, когда поток ожидает объект ядра в alertable-режиме.

39. Понятие элемента работы (Work Item). Назначение элементов работы. Операции с элементами работы. Очереди элементов работы. Обслуживание элементов работы.

- Механизм выполнения асинхронных действий в контексте системного потока на уровне приоритета PASSIVE_LEVEL.
- Зачем? Надо что-то посмотреть, почитать. Обработчик – нельзя, должен быть коротким. DPC – нельзя, вызыв-ся в контексте случ. Потока и может не быть парв читать что-то. APC – тогдав контексте какого потока читать? Поэтому нужен эл-т работы. Обраб-ся в отдельном потоке, созд-ся от имени с-мы, а ей можно всё
- Представляется переменной типа IO_WORKITEM. Создается вызовом IoAllocateWorkItem(), освобождается вызовом IoFreeWorkItem().
- Если драйвер сам выделяет место под структуру IO_WORKITEM, память должна быть не подкачиваемой (non-paged), и драйвер должен ее инициализировать и де-инициализировать вызовами IoInitializeWorkItem() и IoUninitializeWorkItem(). Размер памяти, требуемой для размещения структуры, возвращает IoSizeofWorkItem().
- Чтобы поставить элемент работы в очередь, вызывается IoQueueWorkItem() или IoQueueWorkItemEx(). Один и тот же элемент нельзя ставить в очередь дважды.
- void WorkItemEx(void* IoObject, void* Context, IO_WORKITEM* WorkItem) – процедура программиста, вызываемая операционной системой для обработки элемента работы. Когда работает процедура, элемент работы изъят из очереди. Его можно опять поставить в очередь.
- Вместо создания элемента работы, драйвер может создать системный поток вызовом PsCreateSystemThread(). Но это затратный способ.

При вызове IoQueueWorkItem указывается очередь:

- DelayedWorkQueue – очередь 7-16 обычных потоков с приоритетом 12 и страничным стеком.
- CriticalWorkQueue – очередь 5-16 потоков реального времени с приоритетом 13 и резидентным стеком.
- HyperCriticalWorkQueue – очередь 1 потока реального времени с приоритетом 15. Применяется для удаления завершенных потоков.



40. Управление памятью в ОС Windows. Менеджер памяти. Виртуальная память процесса. Управление памятью в пользовательском режиме. Страничная виртуальная память. Куча (свалка, heap). Проецирование файлов в память.

Менеджер памяти (Memory Manager) выполняет две задачи:
-Отображение виртуальных адресов в физические.
-Страничная организация памяти с отображением страниц на диск.

Аппаратная поддержка: В процессоре имеется Memory Management Unit (MMU) – устройство, выполняющее трансляцию виртуальных адресов в физические.

Виртуальная память процесса: От 2 ГБ до 2 ТБ. Кратна 64 КБ – гранулярность памяти пользовательского режима. Информацию о гранулярности можно получить с помощью **GetSystemInfo**().

Часть виртуальной памяти процесса, которая находится резидентно в физической памяти, называется рабочим набором – Working Set. Диапазон рабочего набора устанавливается функцией **SetProcessWorkingSetSize**(). Стандартный минимальный рабочий набор – 50 страниц по 4 КБ (200 КБ), стандартный максимальный рабочий набор – 345 страниц по 4 КБ (1380 КБ). Без него невозможно на уровне ядра вып-ть даже перекл-е потоков, т.к. для этого тоже нужна память, атк что нужна память, которая там сразу будет

Конфигурация менеджера памяти в реестре:
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management

Страничная виртуальная память:
Выделение: **VirtualAlloc**(), **VirtualAllocEx**(), **VirtualAllocExNuma**() – на опр. Numa-узле, **VirtualFree**(), **VirtualFreeEx**(). Гранулярность в user mode – 64 КБ.
Защита страниц: **VirtualProtect**(), **VirtualProtectEx**().
Фиксация страниц в физической памяти: **VirtualLock**(), **VirtualUnlock**() - Стр-ца не вытесняется на диск. Так лучше не работать, т.к может не остаься страниц для свопинга, но иногда надо
Информация: **VirtualQuery**(), **VirtualQueryEx**().

Куча (свалка) – Heap:
Куча - это достаточно большой непрерывный участок памяти, из которого выделяются небольшие блоки. Для того чтобы куча могла функционировать, в операционную систему был включен так называемый *менеджер кучи. Менеджер кучи* - это специальный механизм, который следит за выделением и освобождением блоков памяти. Для каждого вновь созданного процесса Windows по умолчанию создает кучу. Все кучи, которые создаются операционной системой, являются потокобезопасными. Следовательно, у программиста есть возможность обращаться к одной куче из разных потоков одновременно.

Создание: **HeapCreate**(), **HeapDestroy**().
Выделение: **HeapAlloc**(), **HeapReAlloc**(), **HeapSize**(), **HeapFree**(). Гранулярность – 8 байтов на x86, 16 байтов на x64.
Информация: **HeapValidate**(), **HeapWalk**(), **HeapQueryInformation**(), **HeapSetInformation**().
Кучи процесса: **GetProcessHeap**() – стандартная куча равная 1 MB, **GetProcessHeaps**() – все кучи процесса.
Heapsize – размер блока, а не всей свалки. Станд. Куча есть сразу при созд-и пр-са, для нее не надо делать heapalloc

Отображение файлов в память – File Mapping:
Объект ядра, описывающий отображение фрагмента файла в диапазон виртуальных адресов, называется разделом (Section Object).
Спроец-ть файл в память = саллоцир-ть его в память, кот. будет свяана с памятью на диске

41. Управление памятью в пользовательском режиме ОС Windows. Оптимизация работы кучи с помощью списков предыстории (Look-aside Lists) и низко-фрагментированной кучи (Low Fragmentation Heap).

Управление памятью в пользовательском режиме
-Страничная виртуальная память:
Выделение: VirtualAlloc(), VirtualAllocEx(), VirtualAllocExNuma(), VirtualFree(), VirtualFreeEx(). Гранулярность в user mode – 64 КБ.
Защита страниц: VirtualProtect(), VirtualProtectEx().
Фиксация страниц в физической памяти: VirtualLock(), VirtualUnlock().
Информация: VirtualQuery(), VirtualQueryEx().
Куча (свалка) – Heap:
- Создание: HeapCreate(), HeapDestroy().
- Выделение: HeapAlloc(), HeapReAlloc(), HeapSize(), HeapFree(). Гранулярность – 8 байтов на x86, 16 байтов на x64.
- Информация: HeapValidate(), HeapWalk(), HeapQueryInformation(), HeapSetInformation().
- Кучи процесса: GetProcessHeap() – стандартная куча равная 1 MB, GetProcessHeaps() – все кучи процесса.
Проецирование файлов в память – File Mapping:
- Объект ядра, описывающий отображение фрагмента файла в диапазон виртуальных адресов, называется разделом (Section Object).
Если нужно выделить память неск. Потокам, она может выделиться для них в одной области, т.е. она становится объектом синхр-и потоков, а это плохо. Поэтому оптимизация

Списки предыстории – Look-aside Lists:
-Применяются менеджером кучи для выделения-освобождения элементов фиксированного размера. В ядре могут явно применяться драйверами.
-Представлены в виде 128 связных списков свободных блоков. Каждый список содержит элементы строго определенного размера – от 8 байтов до 1 КБ на x86 и от 16 байтов до 2 КБ на x64.
-Когда блок памяти освобождается, он помещается в список предыстории, соответствующий его размеру. Затем, если запрашивается блок памяти такого же размера, он берется из списка предыстории методом LIFO. Для организации списка предыстории используются функции InterlockedPushEntrySList() и InterlockedPopEntrySList().
-Раз в секунду ОС уменьшает глубину списков предыстории с помощью функции ядра KiAdjustLookasideDepth().

Низко-фрагментированная куча – Low Fragmentation Heap:
-Включается с помощью HeapSetInformation().
-Уменьшает фрагментацию памяти за счет хранения в списках предыстории элементов одного размера вместе.
-Улучшает масштабируемость на многопроцессорных системах путем поддержания количества внутренних структур данных равным количеству процессоров в системе, умноженному на 2.

Можно зайти в список посмотреть, может соседние блоки сливаются в 1, т.е. может вообще вся память свободна. ОС раз в сек это делает. Большая вер-ть, что потоки попросят память из разных списков, т.е уменьш-е вер-ти созд-я общей памяти для потоков

Списки это не оптимизация, это память так устроена. А вот lfh уже можно вкл/выкл. С этим мех-мом все свободные блоки будут лежать подряд, без него при большом кол-ве освоб-й/выдел-й накапливается фрагментация

42. Структура виртуальной памяти в ОС Windows. Виды страниц. Состояния страниц. Структура виртуального адреса. Трансляция виртуального адреса в физический. Кэширование виртуальных адресов.

Виды и состояния страниц

Виртуальная память состоит из двух типов страниц: Малые страницы – 4 КБ. Большие страницы – 4 МБ на x86 или 2 МБ на x64. Большие страницы используются для Ntoskrnl.exe, Hal.dll, данных ядра, описывающих резидентную память ядра и состояние физических страниц памяти.

При вызове VirtualAlloc() можно указать флаг MEM_LARGE_PAGE. Для всей страницы применяется единый режим защиты и блокировки памяти.

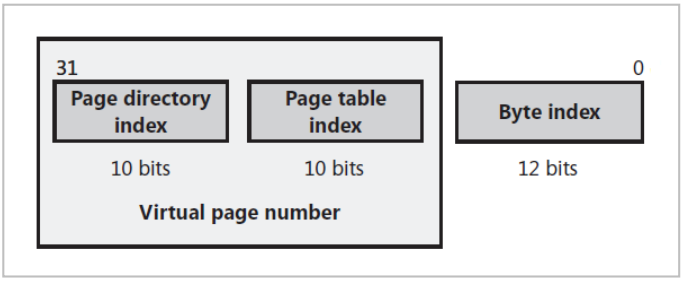
Страницы в виртуальном адресном пространстве процесса могут быть свободными (free, обращение к ней приведёт к сбою), зарезервированными (reserved), переданными: подтвержденными (committed) или общими (shareable). Переданными - Подтвержденными и общими - являются страницы, при обращении к которым в итоге происходит преобразование с указанием настоящих страниц в физической памяти. Подтвержденные страницы называются также закрытыми (private) — это название отражает тот факт, что они, в отличие от общих страниц, не могут использоваться совместно с другими процессами (а могут, разумеется, использоваться только одним процессом). Зарезервирована в таблице страниц – обращ-е к адресу в ней мб обработано с-мой, кот. передаст пр-су реальную стр-цу вирт. Памяти. Для резерв-я непрер. Диапазона вирт. Адресов. Пример – стек потока, он состоит из 1 переданной в физ. Память стр-цы и набора зарез-х стр-ц, т.е. при обращ-и к ним ОС будет подгружать их в физ память

Закрытые страницы выделяются с помощью Windows-функций VirtualAlloc, VirtualAllocEx и VirtualAllocExNuma. Эти функции позволяют программному потоку резервировать адресное пространство, а затем подтверждать части зарезервированного пространства.

Если к подтвержденным (закрытым) страницам до этого еще не было обращений, они создаются во время первого обращения в качестве страниц, подлежащих заполнению нулевыми байтами. Закрытые (подтвержденные) страницы могут быть позже автоматически записаны операционной системой в страничный файл (файл подкачки), если это потребуется для удовлетворения спроса.

При первом обращении к общей странице со стороны какого-нибудь процесса она считывается из связанного отображаемого файла (если только раздел не связан с файлом подкачки, тогда страница создается заполненной нулевыми байтами). Позже, если она все еще находится в физической памяти, то есть является резидентной (resident), при повторном и последующих обращениях процессов можно просто использовать то же самое содержимое страницы, которое уже находится в памяти. Общие страницы могут также заранее извлекаться из памяти самой системой.

Структура виртуального адреса



Номер таблицы страниц в каталоге таблиц (Page directory index - Page Directory Entry).
Номер страницы в таблице страниц (Page table index - Page Table Entry).
Смещение в странице – Byte index.

Трансляция виртуального адреса в физический

Трансляция виртуального адреса – это определение реального (физического) расположение ячейки памяти с данным виртуальным адресом, т. е. преобразование виртуального адреса в физический.
Информация о соответствии виртуальных адресов физическим хранится в таблицах страниц. В системе для каждого процесса поддерживается множество записей о страницах:
Таблицы страниц хранятся в виртуальной памяти. Информация о расположении каждой из таблиц страниц находится в каталоге страниц (Page Directory), единственном для процесса. Записи этого каталога называются PDE (Page Directory Entry). Таким образом, процесс трансляции является двухступенчатым: сначала по виртуальному адресу определяется запись PDE в каталоге страниц, затем поэтой записи находится соответствующая таблица страниц, запись PTE которой указывает на требуемую страницу в физической памяти.

Кэширование виртуальных адресов

Буфер ассоциативной трансляции (TLB) — это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти. TLB используется всеми современными процессорами с поддержкой страничной организации памяти. TLB содержит фиксированный набор записей (от 8 до 4096) и является ассоциативной памятью. Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти.

43. Управление памятью в режиме ядра ОС Windows. Пулы памяти. Выделение и освобождение памяти в пулах памяти. Структура описателя пула памяти. Доступ к описателям пулов памяти на однопроцессорной и многопроцессорной системах.

Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Существуют два типа пулов памяти: Пул резидентной памяти (Non-Paged Pool), в ядре один. Пул страничной памяти (Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пула. Столько надо, что || выделять память и не требовалась синхр-я

Дополнительно операционная система создает и поддерживает:

1. Страничный пул сеансовой памяти (Session Pool).
2. Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти.
3. Пул резидентной памяти, защищенный от исполнения кода (No-Execute Non-Paged Pool – NX Pool).

Начиная с Windows 8, все драйверы должны держать резидентные данные именно в этом пуле.

Выделение памяти в пуле:

void* **ExAllocatePoolWithTag**(POOL_TYPE PoolType,
SIZE_T NumberOfBytes, ULONG Tag);
ExAllocatePoolWithQuota(), **ExAllocatePoolWithQuotaTag**(), **ExAllocatePool**(),
ExAllocatePoolWithTagPriority().

Тегирование блоков памяти в пуле: 4 байта – тег драйвера, например тег Ntfs хранится как строка "sftN. Служит информативным целям

Освоб-е: void ExFreePoolWithTag(void* P, ULONG Tag); ExFreePool().

Каждый пул описывается **структурой POOL_DESCRIPTOR** (Windows 7):

```
struct POOL_DESCRIPTOR
{
   POOL_TYPE PoolType;
   KGUARDED_MUTEX PagedLock;
   ULONG NonPagedLock;
   LONG RunningAllocs;
   LONG RunningDeAllocs;
   LONG TotalBigPages;
   LONG ThreadsProcessingDereferrals;
   ULONG TotalBytes;
   ULONG PoolIndex;
   LONG TotalPages;
   VOID** PendingFrees;
   LONG PendingFreeDepth;
   LIST_ENTRY ListHeads[512];
};
```

Доступ к описателям пулов на однопроцессорной системе: Переменная nt!PoolVector хранит массив указателей на описатели пулов. Первый указывает на описатель резидентного пула. Остальные указывают на описатели страничных пулов. Доступ к ним можно получить через переменную nt!ExpPagedPoolDescriptor. Это массив из указателей на описатели страничных пулов. Количество страничных пулов хранится в переменной nt!ExpNumberOfPagedPools.

Доступ к описателям пулов на многопроцессорной системе:

Каждый NUMA-узел описывается структурой KNODE, в которой хранятся указатели на свой резидентный пул и свой страничный пул NUMA-узла. Указатель на структуру KNODE можно получить из массива nt!KeNodeBlock, в котором хранятся указатели на все KNODE-структуры NUMA-узлов.

Указатели на описатели резидентных пулов всех NUMA-узлов хранятся в массиве nt!ExpNonPagedPoolDescriptor. Количество всех резидентных пулов в системе определяется переменной nt!ExpNumberOfNonPagedPools.

Указатели на описатели страничных пулов всех NUMA-узлов хранятся в массиве nt!ExpPagedPoolDescriptor (по одному на NUMA-узел плюс один). Количество всех страничных пулов в системе определяется переменной nt!ExpNumberOfPagedPools.

44. Пулы памяти ОС Windows. Пул подкачиваемой памяти, пул неподкачиваемой памяти, пул сессии, особый пул. Тегирование пулов. Структура данных пула. Выделение и освобождение памяти в пулах памяти. Организация списков свободных блоков в пуле памяти. Заголовок блока пула памяти.

Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Существуют два типа пулов памяти:

1. Пул резидентной памяти (пул не подкачиваемой памяти, Non-Paged Pool), в ядре один. Ядро и драйвера устройств используют пул не подкачиваемой памяти для хранения информации, к которой можно получить доступ, если система не может обработать paged fault.
2. Пул страничной памяти (пул подкачиваемой памяти, Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пула. Пул подкачиваемой памяти получил своё название из-за того, что Windows может записывать хранимую информацию в файл подкачки, тем самым позволяя повторно использовать занимаемую физическую память. Для пользовательского режима виртуальной памяти, когда драйвер или система ссылается на пул подкачиваемой памяти, который находится в файле подкачки, операция вызывает прерывание и менеджер памяти читает информацию назад в физическую память.

Дополнительно операционная система создает и поддерживает:

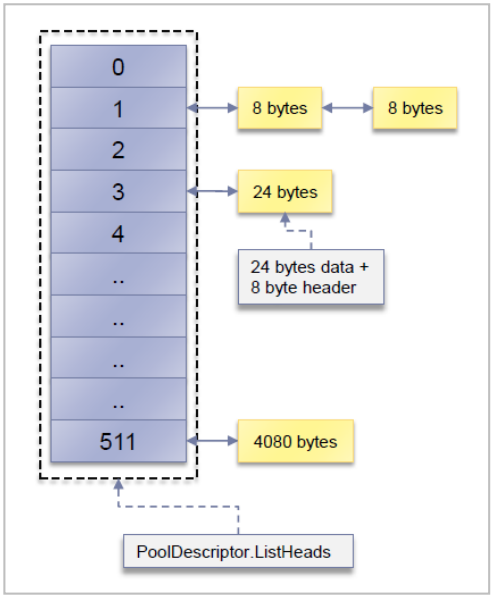
1. Страничный пул сеансовой памяти (Session Pool). Особый страничный пул, в котором располагаются данные сессии.
2. Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти. Страничный/не страничный с возможностью обнаружения нарушения целостности данных (используется для отладки).
3. Пул резидентной памяти, защищенный от исполнения кода (No-Execute Non-Paged Pool – NX Pool). Начиная с Windows 8, все драйверы должны держать резидентные данные именно в этом пуле.

Выделение памяти в пуле:

void* **ExAllocatePoolWithTag**(POOL_TYPE PoolType,
SIZE_T NumberOfBytes, ULONG Tag);
ExAllocatePoolWithQuota(), **ExAllocatePoolWithQuotaTag**(), **ExAllocatePool**(), **ExAllocatePoolWithTagPriority**().

Освоб-е: void ExFreePoolWithTag(void* P, ULONG Tag); ExFreePool().

Тегирование блоков памяти в пуле: 4 байта – тег драйвера, например тег Ntfs хранится как строка "sftN. Служит информативным целям. Tag – тэг пула, использующийся для выделенной памяти. Представляет из себя строковый литерал не более 4 символов, заключённых в одиночные кавычки. Например ‘Tag1’, но определяется она обычно в обратном порядке, т. е. ‘1gaT’. Каждый ASCII символ, входящий в тэг должен иметь значение от 0x20 (пробел) до 0x126 (тильда)..



В описателе пула содержится массив ListHeads: Это 512 двусвязных списков свободных блоков. В каждом списке находятся блоки строго определенного размера от 8 до 4088 байтов с шагом 8 байтов (полезный размер от 0 до 4080). Гранулярность определяется наличием заголовка POOL_HEADER.

Каждый блок памяти имеет заголовок POOL_HEADER, в котором содержится следующая информация:

PreviousSize – размер предыдущего блока.

PoolIndex – индекс пула в массиве описателей, которому блок принадлежит.

BlockSize – размер блока: (NumberOfBytes + 0xF) >> 3 на x86 или (NumberOfBytes + 0x1F) >> 4 на x64.

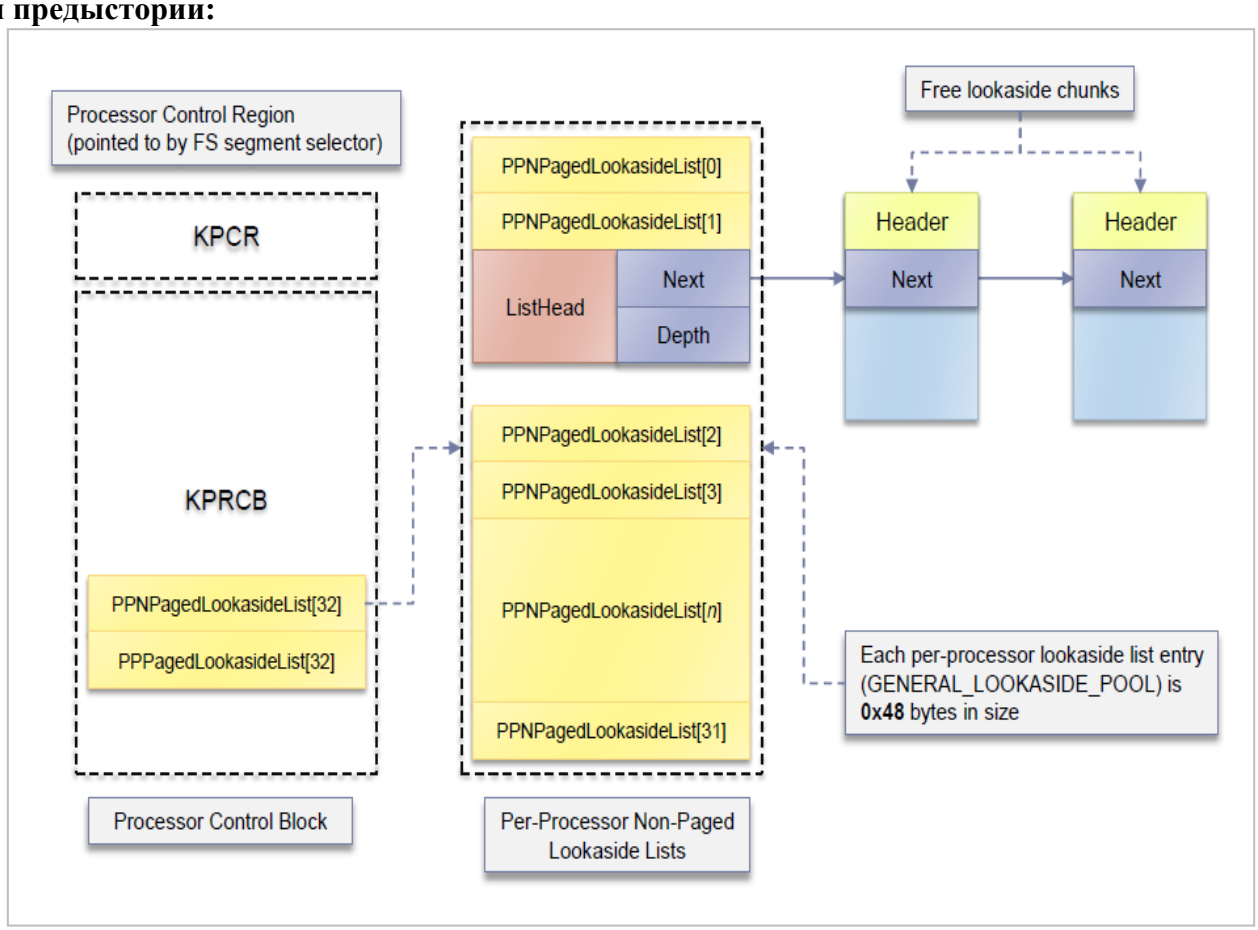
PoolType – тип пула (резидентный, страничный, сеансовый, т.д.).

PoolTag – тег драйвера, выделившего блок.

ProcessBilled – указатель на процесс (структуру EPROCESS), из квоты которого выделен блок (только на x64).

45. Управление памятью в режиме ядра ОС Windows. Оптимизация использования оперативной памяти с помощью списков предыстории – Look-aside Lists.

Списки предыстории:



На каждый процессор создается набор списков предыстории:

1. 32 списка предыстории на процессор.
2. В каждом списке – свободные блоки строго определенного размера

от 8 до 256 байтов с гранулярностью 8 (x86).

Задача – сделать так, чтобы пр-соры работыли независимо др. от др.. 32 списка для страничной памяти, 32 для нестраничной. Тогда каждый пр-сор может работать, не синхр-ясь с другими

Функции работы со списками предыстории:
ExAllocateFromPagedLookasideList(), **ExInitializePagedLookasideList**(),
ExFreeToPagedLookasideList(),
ExDeletePagedLookasideList(), **ExAllocateFromNPagedLookasideList**(),
ExInitializeNPagedLookasideList(), **ExFreeToNPagedLookasideList**(),
ExDeleteNPagedLookasideList().

В блоке состояния процессора KPRCB хранятся 16 специальных списков предыстории. Специальные списки предыстории применяются для:

-Информации о создании объектов.

-Пакетов ввода-вывода (I/O Request Packet – IRP), применяемых для управления драйверами.

-Таблиц описания памяти (Memory Descriptor List – MDL), применяемых для обмена данными на аппаратном уровне.

Для каждого сеанса в MM_SESSION_SPACE хранятся 25 сеансовых списков предыстории.

46. Представление объекта ядра в памяти. Менеджер объектов.

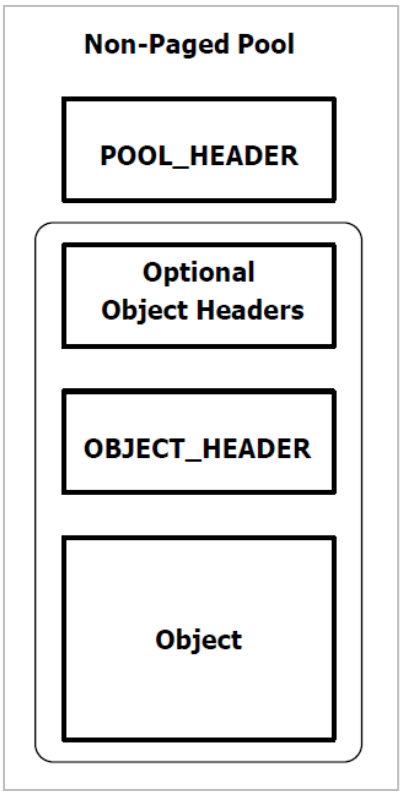
Представление объектов ядра в памяти:

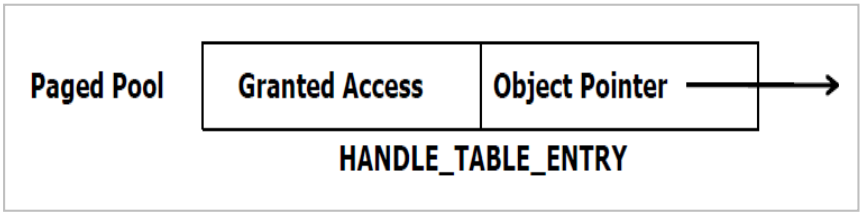
- Таблица указателей на объекты ядра размещается в страничном пуле. Каждый элемент таблицы описывается структурой HANDLE_TABLE_ENTRY, в которой содержится режим доступа к объекту и указатель на объект.
- Объект хранится в резидентном пуле.
- В заголовке объекта хранится указатель на дескриптор защиты объекта, размещающийся в страничном пуле.
- Заголовок блока резидентного пула содержит тег, в котором закодирован тип объекта. Например, у файловых объектов тег называется "File". Это удобно при отладке.

(Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Пул резидентной памяти (Non-Paged Pool), в ядре один.

Пул страничной памяти (Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пул





Разница их в том что страничный состоит из вирутальной памяти которая мб выгружена в/из системы а резидентный пул состоит из адрессов виртуальной памяти, которые гарантированно находятся в физич. всё время, пока выделен соответсвующий объект ядра)

Менеджер объектов.

Так как структура всех объектов одинакова (объект имеет стандартный заголовок и атрибуты), в ОС Windows есть менеджер объектов, который осуществляет поддержку всех объектов. Заголовок объекта включает в себя имя этого объекта, так что другие процессы могут ссылаться на него по имени, а также дескриптор защиты, с помощью которого менеджер объектов может контролировать какие процессы получили доступ к системному ресурсу.

Задачи, выполняемые менеджером объектов:

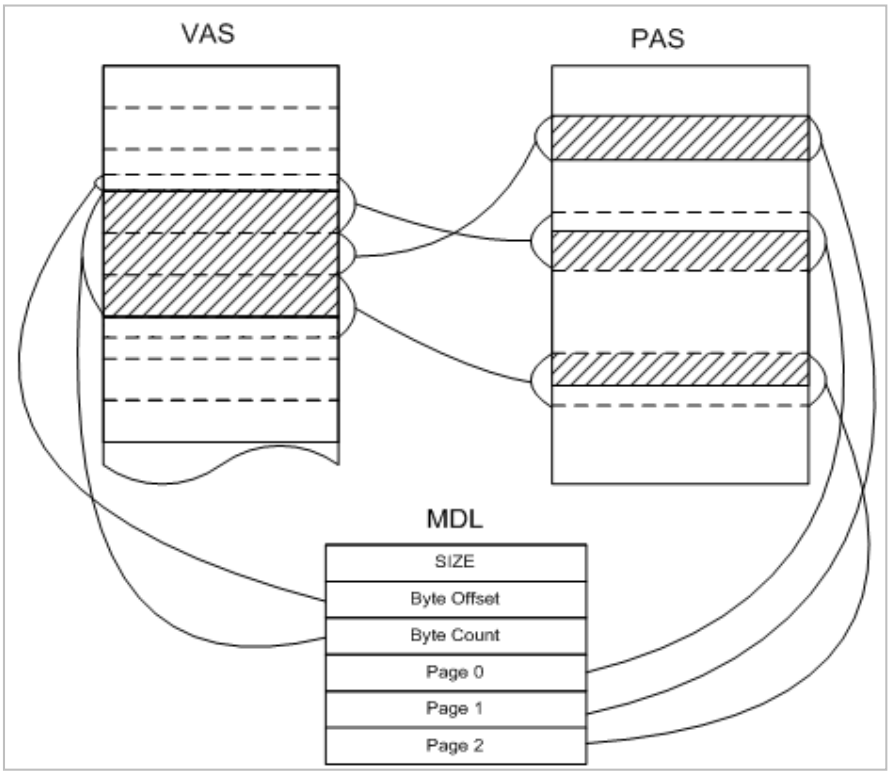
- Создание объектов
- Проверка того, что процесс имеет право на использование объекта
- Создание доступа к объекту ядра и возвращение их к вызывающему
- Поддержание квоты ресурсов
- Закрытие доступа к объектам ядра

47. Фиксация данных в физической памяти ОС Windows. Таблица описания памяти (MDL) и ее использование.

Драйверам необходимо фиксировать данные в физической памяти, чтобы работать с ними на высоких уровнях прерываний. Способы получения физической памяти:
Выделить физически непрерывный блок в резидентном пуле. Дорого!
Выделить физически прерывающийся блок в резидентном пуле.
Выделить блок в страничном пуле и зафиксировать его страницы в физической памяти.
Создать таблицу описания памяти, которая отображает непрерывный блок виртуальной памяти в прерывающийся блок физической памяти. Таблица описания памяти называется Memory Descriptor List (MDL).
Функции для работы с физической памятью:

MmAllocateContiguousMemory(), **MmGetPhysicalAddress**(), **MmLockPageableCodeSection**(), **MmLockPageableDataSection**(), **MmLockPageableSectionByHandle**(), **MmUnlockPageableImageSection**(), **MmPageEntireDriver**(), **MmResetDriverPaging**(),
MmMapIoSpace().

Таблица описания памяти (Memory Descriptor List – MDL) - Структура данных, описывающая отображение буфера виртуальной памяти (Virtual Address Space) в физическую память (Physical Address Space).



данных, описывающая отображение буфера виртуальной памяти (Virtual Address Space) в физическую память (Physical Address Space).
Заголовок MDL, за которым следует массив номеров страниц:

```
struct MDL
{
PMDL Next;
SHORT Size;   SHORT MdlFlags;    PEPROCESS Process;
PVOID MappedSystemVa;  PVOID StartVa;
ULONG ByteCount;  ULONG ByteOffset;
};
MDL* IoAllocateMdl(PVOID VirtualAddress, ULONG Length,
bool SecondaryBuffer, bool ChargeQuota, PIRP Irp);
IoFreeMdl(), IoBuildPartialMdl(),
MmInitializeMdl(), MmSizeOfMdl(), MmBuildMdlForNonPagedPool(),
MmGetMdlVirtualAddress(), MmGetMdlByteCount(), MmGetMdlByteOffset(), MmGetSystemAddressForMdlSafe(),
MmProbeAndLockPages(), MmUnlockPages(), MmMapLockedPages(), MmMapLockedPagesSpecifyCache(),
MmUnmapLockedPages()
```

48. Понятие драйвера ОС Windows. Виды драйверов. Типы драйверов в режиме ядра. Точки входа в драйвер.

В Windows существуют два вида драйверов:
Драйвер режима ядра (kernel-mode driver). Такой драйвер существует в любой версии Windows. Поскольку он подчиняется модели драйверов ядра (Windows Driver Model), его еще называют WDM-драйвер. Правильно написанный WDM-драйвер совместим на уровне исходного кода со всеми версиями ОС. Он имеет деление по типам (см. ниже).
Драйвер пользовательского режима (user-mode driver). Появился, начиная с Windows Vista. Разрабатывается с применением библиотеки Windows Driver Framework (WDF).
Типы драйвера режима ядра:
Драйвер файловой системы (NTFS, FAT, CDFS)
Функциональный драйвер – Functional Driver. Существуют драйверы для классов устройств – Class Drivers. Они предоставляют интерфейсы для расширяющих драйверов – Miniclass Drivers (Minidrivers). Пара Class-Minidriver соответствует полноценному Functional Driver.
Фильтрующий драйвер – Filter Driver. Обеспечивает фильтрацию I/O-запросов между шинным драйвером, функциональным драйвером, драйвером файловой системы.
Шинный драйвер – Bus Driver. Обслуживает физическое устройство с шинной архитектурой (SCSI, PCI, parallel ports, serial ports, i8042 ports).

Объекты в драйвере

DRIVER_OBJECT
Объект, описывающий драйвер. Соответствует программному модулю драйвера. Содержит список создаваемых драйвером устройств.
DEVICE_OBJECT
Контролируемое драйвером физическое или логическое устройство. Содержит указатель на объект, описывающий драйвер. Входит в список устройств драйвера.
FILE_OBJECT
Открытый на устройстве файл. Содержит указатель на устройство.

Главная точка входа в драйвер – DriverEntry:
- NTSTATUS DriverEntry(DRIVER_OBJECT* DriverObject,
UNICODE_STRING* RegistryPath);
- \Registry\Machine\System\CurrentControlSet\Services\DriverName
DriverEntry регистрирует точки входа в драйвер:
- DriverObject->DriverUnload = XxxUnload;
- DriverObject->DriverStartIo = XxxStartIo; // optional
- DriverObject->DriverExtension->AddDevice = XxxAddDevice;
- DriverObject->MajorFunction[IRP_MJ_PNP] = XxxDispatchPnp;
- DriverObject->MajorFunction[IRP_MJ_POWER] = XxxDispatchPower;
- Другие стандартные точки входа (ISR, IoCompletion) регистрируются с помощью предназначенных для этого функций ядра.
DriverEntry выполняет дополнительные действия:
- Вызывает IoAllocateDriverObjectExtension, если нужно хранить дополнительные данные, ассоциированные с драйвером;
- Вызывает IoRegisterDriverReinitialization(…, XxxReinitialize,…) или IoRegisterBootDriverReinitialization(…, XxxReinitialize,…), если после вызова DriverEntry у всех драйверов следует продолжить инициализацию.

49. Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Структура и взаимосвязь объектов.

Объект DRIVER_OBJECT:

Представляет загруженный в память драйвер. Создается в единственном экземпляре в момент загрузки модуля драйвера в память операционной системой. Уничтожается в момент выгрузки модуля драйвера из памяти. Передается в главную точку входа DriverEntry и процедуру XxxAddDevice. Хранит состояние, общее для всех обслуживаемых драйвером устройств.

Содержит:

Имя драйвера в структуре имен операционной системы. Начальный адрес и размер драйвера в памяти.Таблицу точек входа в драйвер.Указатель на область расширенных данных драйвера, в которой хранится точка входа в процедуру XxxAddDevice.Список созданных драйвером объектов DEVICE_OBJECT, представляющих обслуживаемые драйвером устройства.

Объект DEVICE_OBJECT:

Представляет физическое или логическое устройство.Создается драйвером с помощью процедуры **IoCreateDevice**(). Уничтожается с помощью процедуры **IoDeleteDevice**().Передается в процедуру XxxAddDevice. Может не иметь имени. Тогда оно автоматически генерируется операционной системой – FILE_AUTOGENERATED_DEVICE_NAME.

Содержит:Фиксированную (DEVICE_OBJECT) и переменную часть данных устройства. Размер и содержимое переменной части определяются драйвером.Указатель на область расширенных данных объекта – DEVOBJ_EXTENSION. Таким образом, расширений получается два.Указатель на владельца – объект драйвера – DriverObject. Указатель на такой же объект устройства в драйвере верхнего уровня – AttachedDevice. Получающийся список образует стек драйверов.Указатель на следующее устройство в списке драйвера – NextDevice.Указатель на очередь I/O-запросов к устройству – DeviceQueue, и текущий запрос к устройству – CurrentIrp.

Объект FILE_OBJECT:

Представляет файл, открытый на устройстве.Создается вызовом **CreateFile**()/**ZwCreateFile**().Удаляется вызовом **CloseHandle**()/**ZwClose**().

Содержит:

Указатель на владельца – объект устройства – DeviceObject.Относительное имя файла, интерпретируемое драйвером устройства или драйвером файловой системы, – FileName.Дополнительные данные, необходимые драйверам для работы с файлом, – FsContext и FsContext2.Указатель на блок параметра тома, устанавливающий соответствие между файловой системой и смонтированным томом на устройстве, – Vpb.Объект синхронизации Event, который блокирует потоки, осуществляющие синхронные запросы к устройству. Обработка запросов выполняется асинхронно.

50. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при открытии файла.

Пакет ввода-вывода – Input-Output Request Packet (IRP) РИС 1

IRP пакет (I/O request packet) — структура данных ядра Windows, обеспечивающая обмен данными между приложениями и драйвером, а также между драйвером и драйвером. Представляет запрос ввода-вывода.
Создается с помощью **IoAllocateIrp**() или **IoMakeAssociatedIrp**() или **IoBuildXxxRequest**(). Память выделяется из списка предыстории в резидентном пуле.
Удаляется вызовом **IoCompleteRequest**().
Диспетчируется драйверу на обработку с помощью **IoCallDriver**().
Структура IRP:

Фиксированную (IRP) и переменную часть в виде массива записей IO_STACK_LOCATION. Количество элементов массива – поле StackCount. На каждый драйвер в стеке драйверов создается отдельная запись IO_STACK_LOCATION.
Объект FILE_OBJECT, с которым осуществляется работа, – Tail.Overlay.OriginalFileObject.
Буфер данных в пользовательской памяти – UserBuffer.
Буфер данных в системной памяти – AssociatedIrp.SystemBuffer.
Соответствующая буферу таблица описания памяти – MdlAddress.
Указатель на поток (ETHREAD), в очереди которого находится IRP, – Tail.Overlay.Thread. Список IRP потока хранится в ETHREAD.IrpList.

Схема обработки пакета ввода-вывода при открытии файла РИС2

1.Подсистема ОС вызывает функцию открытия файла в ядре. Эта функция реализована в менеджере ввода-вывода.
2.Менеджер ввода-вывода обращается к менеджеру объектов, чтобы по имени файла создать FILE_OBJECT. При этом осуществляется проверка прав пользователя на обращение к файлу.
3.При открытии файл может находиться на еще не смонтированном томе. В таком случае открытие файла приостанавливается, выполняется монтирование тома на внешнем устройстве и обработка продолжается.
4.Менеджер ввода-вывода создает и инициализирует IRP-пакет с помощью IoAllocateIrp(). В IRP-пакете инициализируется IO_STACK_LOCATION верхнего драйвера в стеке драйверов.
5.Менеджер ввода-вывода вызывает процедуру XxxDispatchCreate() верхнего драйвера. Процедура драйвера вызывает IoGetCurrentIrpStackLocation(), чтобы получить доступ к параметрам запроса. Она проверяет, не кэширован ли файл. Если нет, то вызывает IoCopyCurrentIrpStackLocationToNext() для создания IO_STACK_LOCATION следующего драйвера в стеке, затем IoSetCompletionRoutine() для получения уведомления о завершении обработки IRP-пакета и вызывает IoCallDriver(), делегируя обработку процедуре YyyDispatchCreate() следующего драйвера в стеке.
6.Каждый драйвер в стеке выполняет свою часть обработки IRP-пакета.
7.Последний драйвер в стеке в своей процедуре YyyDispatchCreate() устанавливает в IRP поле IoStatus и вызывает у менеджера ввода-вывода процедуру IoCompleteRequest(), чтобы завершить обработку IRP-пакета. Она проходит в IRP по массиву записей IO_STACK_LOCATION и в каждой вызывает процедуру CompletionRoutine (указывает на XxxIoCompletion() драйвера).
8.Менеджер ввода-вывода проверяет в IRP.IoStatus и копирует соответствующий код возврата в адресное пространство подсистемы ОС (пользовательского процесса).
9.Менеджер ввода-вывода удаляет IRP-пакет с помощью IoFreeIrp().
10.В адресном пространстве пользователя создается описатель для FILE_OBJECT и возвращается подсистеме ОС как результат открытия файла. В случае ошибки возвращается ее код.

51. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при выполнении чтения-записи файла.

Понятие пакета IRP и его структура

Берём из предыдущего вопроса…

Схема обработки IRP при выполнении чтения-записи файла

1.Менеджер ввода-вывода обращается к драйверу файловой системы с IRP-пакетом, созданным для выполнения чтения-записи файла. Драйвер обращается к своей записи IO_STACK_LOCATION и определяет, какую именно операцию он должен выполнить.
2.Драйвер файловой системы для выполнения операции с файлом может создавать свои IRP с помощью IoAllocateIrp(). Или же он может в уже имеющемся IRP сформировать IO_STACK_LOCATION для драйвера более низкого уровня с помощью IoGetNextIrpStackLocation().
3.Если драйвер создает собственные IRP-пакеты, он должен зарегистрировать в них свою процедуру ZzzIoCompletion(), которая выполнит удаление IRP-пакетов после обработки драйверами нижнего уровня. За удаление своих IRP каждый драйвер отвечает сам. Менеджер ввода-вывода отвечает за удаление своего IRP, созданного для выполнения ввода-вывода.
Драйвер файловой системы устанавливает в IO_STACK_LOCATION указатель CompletionRoutine на свою процедуру XxxIoCompletion(), формирует IO_STACK_LOCATION для драйвера более низкого уровня с помощью IoGetNextIrpStackLocation(), вписывая нужные значения параметров, и обращается к драйверу более низкого уровня с помощью IoCallDriver().
4.Управление передается драйверу устройства процедуре YyyDispatchRead/Write(), зарегистрированной в объекте DRIVER_OBJECT под номером IRP_MJ_XXX. Драйвер устройства не может выполнить операцию ввода-вывода в синхронном режиме. Он помечает IRP-пакет с помощью IoMarkIrpPending() как требующий ожидания обработки или передачи другой процедуре YyyDispatch().
5.Менеджер ввода-вывода получает информацию, что драйвер устройства занят, и ставит IRP в очередь к объекту DEVICE_OBJECT драйвера.
6.Когда устройство освобождается, в драйвере устройства вызывается процедура обработки прерываний (ISR). Она обнаруживает IRP в очереди к устройству и с помощью IoRequestDpc() создает DPC-процедуру для обработки IRP на более низком уровне приоритета прерываний.
7.DPC-процедура с помощью IoStartNextPacket() извлекает из очереди IRP и выполняет ввод-вывод. Наконец, она устанавливает статус-код в IRP и вызывает IoCompleteRequest().
8.В каждом драйвере в стеке вызывается процедура завершения XxxIoCompletion(). В драйвере файловой системы она проверяет статус-код и либо повторяет запрос (в случае сбоя), либо завершает его удалением всех собственных IRP (если они были). В конце, IRP-пакет оказывается в распоряжении менеджера ввода-вывода, который возвращает вызывающему потоку результат в виде NTSTATUS.

52. Перехват API-вызовов ОС Windows в пользовательском режиме. Внедрение DLL с помощью реестра. Внедрение DLL с помощью ловушек. Внедрение DLL с помощью дистанционного потока.

Перехват API-вызовов ОС Windows в пользовательском режиме.
Задача – изменить поведение окна:
HWND hwnd = FindWindow(ClassName, WindowName);
SetClassLongPtr(hwnd, GWLP_WNDPROC, MyWindowProc);
Изменяемый оконный класс может находиться в адресном пространстве другого процесса, и адрес процедуры MyWindowProc будет не валиден.
Внедрение DLL с помощью реестра:
Зарегистрировать DLL в реестре (имя не должно содержать пробелы):
HKLM\Software\Microsoft\Windows_NT\CurrentVersion\Windows\AppInit_DLLs
Выполнить API-перехват в DllMain (reason == DLL_PROCESS_ATTACH). Функции Kernel32.dll можно вызывать смело. С вызовами функций из других DLL могут быть проблемы.

Внедрение DLL с помощью ловушек:
HHOOK **SetWindowsHookEx**(int idHook, HOOKPROC lpfn,
INSTANCE hMod, DWORD dwThreadId); **UnhookWindowsHookEx**().
Процедура ловушки:
LRESULT **MyHookProc**(int code, WPARAM wParam, LPARAM lParam);
code: если HC_ACTION, надо обработать, если меньше нуля, – вызвать:
LRESULT **CallNextHookEx**(HHOOK hhook, int code, WPARAM wParam, LPARAM lParam);

Внедрение DLL с помощью дистанционного потока:
HANDLE **CreateRemoteThread**(HANDLE hProcess, SECURITY_ATTRIBUTES* securityAttributes,
DWORD dwStackSize, THREAD_START_ROUTTNE* startAddress, void* parameter,
DWORD dwCreationFlags, DWORD* pThreadId);
DWORD **ThreadProc**(void* parameter);
HINSTANCE **LoadLibrary**(PCTSTR fileName);
void* p = GetProcAddress(GetModuleHandle("Kernel32"), "LoadLibraryW");
Передача данных в дистанционный поток:
void* **VirtualAllocEx** (HANDLE hProcess, void* lpAddress, SIZE_T dwSize, DWORD flAllocationType,
DWORD flProtect);
bool **VirtualFreeEx**(HANDLE hProcess, void* lpAddress, SIZE_T dwSize, DWORD dwFreeType);
bool **WriteProcessMemory**(HANDLE hProcess, void* lpBaseAddress,
const void* lpBuffer, SIZE_T nSize, SIZE_T* lpNumberOfBytesWritten);
bool **ReadProcessMemory**(HANDLE hProcess, void* lpBaseAddress,
const void* lpBuffer, SIZE_T nSize, SIZE_T* lpNumberOfBytesRead);

53. Перехват API-вызовов ОС Windows в пользовательском режиме. Замена адреса в таблице импорта. Перехват в точке входа в процедуру с помощью подмены начальных инструкций (Microsoft Detours).

Задача – изменить поведение окна:

HWND hwnd = FindWindow(ClassName, WindowName);

SetClassLongPtr(hwnd, GWLP_WNDPROC, MyWindowProc);

Изменяемый оконный класс может находиться в адресном пространстве другого процесса, и адрес процедуры MyWindowProc будет не валиден.

Замена адреса в таблице импорта:

void* **ImageDirectoryEntryToDataEx**(void* Base, // hModule
bool MappedAsImage, USHORT DirectoryEntry, ULONG* Size, IMAGE_SECTION_HEADER** FoundHeader);

DirectoryEntry: IMAGE_DIRECTORY_ENTRY_IMPORT

Перехват в точке входа в процедуру с помощью подмены начальных инструкций

Библиотека Detours облегчает перехват вызовов функций. Перехватывающий код применяется динамически во время выполнения. Detours заменяет первые несколько инструкций целевой функции на безусловный переход к предоставленной пользователем функции. Инструкции целевой функции сохраняются в функции-трамплине. Трамплин состоит из инструкций, удаленных из целевой функции и безусловного перехода к остальной части целевой функции. Когда выполнение достигает целевую функцию, управление передается непосредственно к пользовательской функции перехвата. Функция перехвата выполняет всю необходимую предварительную обработку. Она может вернуть управление исходной функции или может вызвать функцию-трамплин, которая вызывает целевую функцию без перехвата. Когда целевая функция завершается, она возвращает управление функции перехвата. Функция перехвата выполняет соответствующие постобработки и возвращает управление исходной функции. На рисунке 1 показан логический поток управления для вызова функции с и без перехвата.

Библиотека Detours перехватывает целевые функции, переписав их двоичный образ в памяти. Для каждой целевой функции, Detours фактически переписывает две функции: целевую функцию и соответствующую функцию-трамплин. Функция-трамплин может быть выделена либо динамически, либо статически. Статически выделенный трамплин всегда вызывает целевую функцию без перехвата. До вставки перехвата, статический трамплин содержит один прыжок к целевой функции. После вставки, трамплин содержит начальные инструкции из целевой функции и прыжка к остальной части целевой функции.

Рисунок 2 показывает вставку перехвата. Для перехвата целевой функции, Detours сперва выделяет память для динамической функции-трамплина (если статического трамплина не предусмотрено), а затем открывает доступ на запись в целевой функции и в трамплине. Начиная с первой инструкции, Detours копирует инструкции из целевой функции в трамплин как минимум 5 байт (достаточно для безусловного оператора перехода). Если целевая функция менее 5 байт, Detours прерывается и возвращает код ошибки. Detours добавляет инструкцию перехода в конец трамплина к первой не скопированной инструкции целевой функции. Detours записывает инструкцию безусловного перехода в функцию перехвата первой инструкцией целевой функции. Наконец, Detours восстанавливает исходные разрешения доступа для страниц целевой функции и трамплина и очищает кэш команд CPU с помощью вызова FlushInstructionCache.

54. Перехват API-вызовов ОС Windows в режиме ядра. Таблица системных функций KeServiceDescriptorTable. Таблица системных функций KeServiceDescriptorTableShadow. Понятие UI-потока. Защита от перехвата (Kernel Patch Protection) в 64-разрядной ОС Windows.

KeServiceDescriptorTable:

Переменная, указывающая на таблицу API-функций ядра.

Экспортируется ядром и видна драйверам.

Номер функции может зависеть от версии ОС.

По номеру функции можно заменить адрес функции в этой таблице.

Таблица защищена от модификации, поэтому перед заменой нужно или отключить бит защиты страницы, или создать доступное для записи отображение таблицы (writable Kernel Virtual Address (KVA) mapping).

KeServiceDescriptorTableShadow:

Переменная, указывающая на таблицы API-функций ядра и Win32k.sys.

Не экспортируется ядром и не видна драйверам. Это осложняет перехват функций работы с окнами и графикой.

UI-потоки содержат указатель на эту таблицу в ETHREAD.Tcb.ServiceTable, но смещение до этого поля отличается в каждой ОС.

UI-поток должен обращаться к драйверу за перехватом UI-функций.

Перехват UI-функций во время загрузки ОС становится проблематичен.

Защита от перехвата – Kernel Patch Protection:

Реализована на 64-разрядной платформе.

Не дает модифицировать:

- GDT – Global Descriptor Table
- IDT – Interrupt Descriptor Table
- MSRs – Model-Specific Registers
- Kernel Service Table

В случае модификации вызывается KeBugCheckEx() с кодом CRITICAL_STRUCTURE_CORRUPTION. При этом стек зачищается, чтобы осложнить реверс-инжиниринг.

Инвестиции в обход механизма Kernel Patch Protection себя не окупают. Microsoft изменяет работу этого механизма в новых обновлениях.

55. Перехват API-вызовов менеджера объектов ОС Windows в режиме ядра.

Установка разрешенного набора callback-процедур для некоторых подсистем ядра:
Object Manager Callbacks
Process Callbacks
Thread Callbacks
Module Load Callbacks
Registry Callbacks
File System Mini-Filters
Win32k.sys такого механизма не имеет
Требования к драйверам, применяющим эти механизмы:
Драйвер должен быть скомпонован с ключом /integritycheck.
Драйвер должен быть подписан сертификатом производителя ПО.
При разработке драйвера должен быть включен режим действия тестовых сертификатов:
C:\> bcdedit.exe –set TESTSIGNING ON
Несколько драйверов могут устанавливать callback-процедуры на конкурентной основе. Для них Windows применяет уровни перехвата, за исключением Process, Thread и Module Load Callbacks.

- Заменяют перехват следующих процедур native API:
  - NtOpenProcess(), NtOpenThread(), NtDuplicateObject() для описателей процессов и потоков.
- Регистрация процедур перехвата:
  - NTSTATUS **ObRegisterCallbacks**(OB_CALLBACK_REGISTRATION* CallBackRegistration, PVOID* RegistrationHandle);
  - void **ObUnRegisterCallbacks**(PVOID RegistrationHandle);
  - struct OB_CALLBACK_REGISTRATION { USHORT Version;
    USHORT OperationRegistrationCount; UNICODE_STRING Altitude; PVOID RegistrationContext; OB_OPERATION_REGISTRATION* OperationRegistration; };
  - struct OB_OPERATION_REGISTRATION {
    POBJECT_TYPE* ObjectType; // PsProcessType или PsThreadType
    OB_OPERATION Operations;
    OB_PRE_OPERATION_CALLBACK* PreOperation;
    OB_POST_OPERATION_CALLBACK* PostOperation; };
  - OB_PREOP_CALLBACK_STATUS **ObjectPreCallback**(PVOID RegistrationContext, OB_PRE_OPERATION_INFORMATION* OperationInfo);
  - void **ObjectPostCallback**(PVOID RegistrationContext, OB_POST_OPERATION_INFORMATION* OperationInfo);

56. Перехват API-вызовов создания и уничтожения процессов и потоков ОС Windows в режиме ядра.

Процедура перехвата создания и уничтожения процесса:

NTSTATUS **PsSetCreateProcessNotifyRoutineEx**(
CREATE_PROCESS_NOTIFY_ROUTINE_EX* NotifyRoutine, bool Remove);
параметр Remove выбирает между регистрацией и удалением.

void **ProcessNotifyEx**(EPROCESS* Process, HANDLE ProcessId, PS_CREATE_NOTIFY_INFO* CreateInfo) – формат NotifyRoutine.

Процедура вызываются на уровне приоритета прерываний
PASSIVE_LEVEL в контексте потока, вызывающего создание описателя.

struct PS_CREATE_NOTIFY_INFO { SIZE_T Size; union { ULONG Flags; struct {
ULONG FileOpenNameAvailable :1; ULONG Reserved :31; }; };
HANDLE ParentProcessId; CLIENT_ID CreatingThreadId;
FILE_OBJECT* FileObject; const UNICODE_STRING* ImageFileName;
const UNICODE_STRING* CommandLine; NTSTATUS CreationStatus; };

Процедура программиста может запретить создание процесса, если установит в поле CreateInfo->CreationStatus ненулевой код ошибки.

ОС позволяет зарегистрировать не более 12 таких процедур перехвата.

Устаревшая процедура перехвата в версиях до Windows Vista:

PsSetCreateProcessNotifyRoutine() – по формату аналогична.

void **ProcessNotify**(HANDLE ParentId, HANDLE ProcessId, bool Create);
Уведомление без возможности запретить создание/удаление процесса.

Процедура перехвата создания и уничтожения потока:

NTSTATUS **PsSetCreateThreadNotifyRoutine**(
CREATE_THREAD_NOTIFY_ROUTINE* NotifyRoutine);

NTSTATUS **PsRemoveCreateThreadNotifyRoutine**(
CREATE_THREAD_NOTIFY_ROUTINE* NotifyRoutine);

void **ThreadNotify**(HANDLE ProcessId, HANDLE ThreadId, bool Create);

Создание и удаление потока отменить нельзя.

Обычно используется драйверами для очистки создаваемых для потоков ресурсов.

ОС позволяет зарегистрировать не более 8 таких процедур перехвата.

57. Перехват операций с реестром в ОС Windows в режиме ядра.

Процедура перехвата операций с реестром Windows:

NTSTATUS **CmRegisterCallbackEx**(EX_CALLBACK_FUNCTION* Function,
const UNICODE_STRING* Altitude, void* Driver, void* Context,
LARGE_INTEGER* Cookie, void* Reserved); **CmRegisterCallback**() устарела.

NTSTATUS **CmUnRegisterCallback**(LARGE_INTEGER Cookie);

NTSTATUS **RegistryCallback**(void* Context, REG_NOTIFY_CLASS Argument1, void* Argument2); // Argument2 – указатель на REG_XXX_INFORMATION.

Процедура перехвата операций с реестром может выполнять:
мониторинг, блокировку (XP и выше) и модификацию (Vista и выше).

Если операция предотвращается с помощью STATUS_CALLBACK_BYPASS, вызывающий поток получает STATUS_SUCCESS. Если операция предотвращается с помощью кода ошибки, поток получает ее код.

В процедуре перехвата при создании ключа реестра можно назначить ключу отдельный контекст, который будет передаваться в процедуру с уведомлениями REG_XXX_KEY_INFORMATION:

NTSTATUS **CmSetCallbackObjectContext**(void* Object,
LARGE_INTEGER* Cookie, void* NewContext, void** OldContext);

При ассоциации контекста с ключом реестра, процедуре перехвата будет послано уведомление об уничтожении ключа.

- Процедура перехвата загрузки (отображения в память) модуля:
  - NTSTATUS **PsSetLoadImageNotifyRoutine**(
    LOAD_IMAGE_NOTIFY_ROUTINE* NotifyRoutine);
  - NTSTATUS **PsRemoveLoadImageNotifyRoutine**(
    LOAD_IMAGE_NOTIFY_ROUTINE* NotifyRoutine);
  - void **ThreadNotify**(UNICODE_STRING* FullImageName, HANDLE ProcessId, IMAGE_INFO* ImageInfo);
  - При загрузке модуля драйвера ProcessId равен NULL.
  - struct IMAGE_INFO { union { ULONG Properties; struct {
    ULONG ImageAddressingMode :8; //code addressing mode
    ULONG SystemModeImage :1; //system mode image
    ULONG ImageMappedToAllPids :1; //mapped in all processes
    ULONG Reserved :22; }; };
    PVOID ImageBase; ULONG ImageSelector; ULONG ImageSize;
    ULONG ImageSectionNumber; };
  - Загрузку модуля отменить нельзя.
  - Обычно используется драйверами для модификации таблицы импорта загружаемого модуля. Не вызывается, если загрузка модуля происходит с атрибутом SEC_IMAGE_NO_EXECUTE.
  - ОС позволяет зарегистрировать не более 8 таких процедур перехвата.

58. Перехват операций с файлами в ОС Windows в режиме ядра. Мини-фильтры файловой системы.

Перехват процедур взаимодействия программ с файловой системой возможен в двух вариантах:

Установка фильтрующего драйвера – File System Filter Driver. Устаревший способ, унаследованный от предыдущих версий Windows (до Win 2000).
Установка фильтрующего мини-драйвера (мини-фильтра) – File System Mini-Filter. Способ основан на запуске компонента Filter Manager (fltmgr.sys) как фильтрующего драйвера, запускающего и контролирующего мини-фильтры.
Filter Manager: Активизируется при загрузке первого же мини-фильтра. Обеспечивает работу множества мини-фильтров. Может загружать и выгружать мини-фильтры без перезагрузки системы.
Устанавливается в стеке драйверов между менеджером ввода-вывода и драйверами файловой системы (NTFS, FAT и др.). Умеет устанавливаться в обхват других фильтрующих драйверов (см. рисунок). При установке мини-фильтров применяет параметр высоты установки (Altitude).
Скрывает сложность модели ввода-вывода на основе IRP-пакетов и предоставляет интерфейс для регистрации функций перехвата. Параметры в функции перехвата приходят разобранными (не IRP).
Предоставляет API и для ядра, и для пользовательского режима.
Mini-Filter: Загружается и управляется как из kernel mode, так и из user mode. Функции режима ядра – **FltXxxYyy**(), функции приложений – **FilterXxx**().
Перехватчики устанавливаются мини-фильтром в режиме ядра.Для каждого тома файловой системы и присоединенного к нему мини-фильтра Filter Manager создает ассоциированный объект, называемый Instance. Мини-фильтр может перехватывать создание и удаление таких объектов.
Загрузка и выгрузка мини-фильтра:

```
NTSTATUS FltLoadFilter(PCUNICODE_STRING FilterName);
NTSTATUS FltUnloadFilter(PCUNICODE_STRING FilterName);
HRESULT FilterLoad(LPCWSTR lpFilterName); // user mode
HRESULT FilterUnload(LPCWSTR lpFilterName); // user mode
```

Регистрация процедур перехвата в мини-фильтре:

```
NTSTATUS FltRegisterFilter(DRIVER_OBJECT* Driver, const FLT_REGISTRATION* Registration, PFLT_FILTER* RetFilter);
NTSTATUS FltStartFiltering(PFLT_FILTER Filter);
void FltUnregisterFilter(PFLT_FILTER Filter);
```

Контексты Filter Manager-а в мини-фильтре:
Если мини-фильтру нужно ассоциировать свои данные с объектами Filter Manager-а, он устанавливает массив перехватчиков на каждый из таких объектов – так называемых контекстов.
Существуют следующие контексты: том (volume), экземпляр мини-фильтра для тома (instance), поток ввода-вывода (stream), описатель потока ввода-вывода (stream handle), секция отображаемого файла (section), транзакция (transaction), файл (file). См. поле ContextType ниже.
Регистрация процедур перехвата операций ввода-вывода в мини-фильтре:

```
FLT_PREOP_CALLBACK_STATUS PreOperationCallback(
FLT_CALLBACK_DATA* Data, const FLT_RELATED_OBJECTS* FltObjects,
PVOID* CompletionContext);
FLT_POSTOP_CALLBACK_STATUS PostOperationCallback(
FLT_CALLBACK_DATA* Data, const FLT_RELATED_OBJECTS* FltObjects,
PVOID CompletionContext, FLT_POST_OPERATION_FLAGS Flags);
struct FLT_RELATED_OBJECTS { const USHORT Size;
   const USHORT TransactionContext; const FLT_FILTER* Filter;
   const FLT_VOLUME* Volume; const FLT_INSTANCE* Instance;
   const FILE_OBJECT* FileObject; const KTRANSACTION* Transaction; };
```

1. Модель программного интерфейса операционной системы Windows. Нотация программного интерфейса. Понятие объекта ядра и описателя объекта ядра операционной системы Windows. Модель архитектуры ОС Windows.
2. Понятие пользовательского режима и режима ядра операционной системы Windows. Модель виртуальной памяти процесса в пользовательском режиме и в режиме ядра операционной системы Windows. Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows. Основные модули ОС Windows.
3. Системный реестр операционной системы Windows. Структура и главные разделы. Точки автозапуска программ. Средства редактирования реестра Windows. Функции работы с реестром из приложения.
4. Понятие окна в ОС Windows. Основные элементы окна. Понятие родительского и дочернего окна. Структура программы с событийным управлением. Минимальная программа для ОС Windows с окном на экране. Создание и отображение окна.
5. Структура программы с событийным управлением. Структура события – оконного сообщения Windows. Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений. Процедуры посылки сообщений. Синхронные и асинхронные сообщения.
6. Ввод данных с манипулятора «мышь». Обработка сообщений мыши. Ввод данных с клавиатуры. Понятие фокуса ввода. Обработка сообщений от клавиатуры.
7. Вывод информации в окно. Механизм перерисовки окна. Понятие области обновления окна. Операции с областью обновления окна.
8. Принципы построения графической подсистемы ОС Windows. Понятие контекста устройства. Вывод графической информации на физическое устройство. Управление цветом. Палитры цветов. Графические инструменты. Рисование геометрических фигур.
9. Растровые изображения. Виды растровых изображений. Значки и курсоры. Способ вывода растровых изображений с эффектом прозрачного фона. Аппаратно-зависимые и аппаратно-независимые растровые изображения. Операции с растровыми изображениями. Вывод растровых изображений.
10. Библиотека работы с двумерной графикой Direct2D. Инициализация библиотеки. Фабрика графических объектов библиотеки Direct2D. Вывод графики средствами библиотеки Direct2D.
11. Вывод текста в ОС Windows. Понятие шрифта. Характеристики шрифта. Понятия физического и логического шрифта. Операции с физическими шрифтами. Операции с логическими шрифтами. Параметры ширины и высоты логического шрифта.
12. Системы координат. Трансформации. Матрица трансформаций. Виды трансформаций и их представление в матрице трансформаций. Преобразования в страничной системе координат. Режимы масштабирования.
13. Понятие ресурсов программ Windows. Виды ресурсов. Операции с ресурсами.
14. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом статического импорта процедур. Соглашения о вызовах процедур DLL-библиотеки. Точка входа-выхода DLL-библиотеки.
15. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом динамический импорта процедур.
16. Понятие динамически-загружаемой библиотеки. Создание в DLL-библиотеке разделяемых между приложениями глобальных данных. Разделы импорта и экспорта DLL-библиотеки. Переадресация вызовов процедур DLL-библиотек к другим DLL-библиотекам. Исключение конфликта версий DLL.
17. Понятие объекта ядра ОС Windows. Виды объектов ядра. Атрибуты защиты объекта ядра. Дескриптор защиты объекта ядра. Создание и удаление объектов ядра.
18. Проецирование файлов в память. Отличие в механизме проецирования файлов в память в ОС Windows и UNIX/Linux. Действия по проецированию файла в память.
19. Современные многопроцессорные архитектуры SMP и NUMA. Многоуровневое кэширование памяти в современных процессорах. Проблема перестановки операций чтения и записи в архитектурах с ослабленной моделью памяти. Способы решения проблемы перестановки операций чтения и записи.
20. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Достоинства и недостатки процессов и потоков. Создание и завершение процесса. Запуск процессов по цепочке.
21. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Создание и завершение потока. Приостановка и возобновление потока. Контекст потока.
22. Понятие пула потоков. Архитектура пула потоков. Операции с потоками при работе с пулом потоков.
23. Распределение процессорного времени между потоками ОС Windows. Механизм приоритетов. Класс приоритета процесса. Относительный уровень приоритета потока. Базовый и динамический приоритеты потока. Операции с приоритетами.
24. Механизмы синхронизации потоков одного и разных процессов в ОС Windows. Обзор и сравнительная характеристика механизмов синхронизации.
25. Синхронизация потоков в пределах одного процесса ОС Windows. Критическая секция. Операции с критической секцией. Атомарные операции.
26. Синхронизация потоков в пределах одного процесса ОС Windows. Ожидаемое условие (монитор Хора). Операции с ожидаемым условием. Пример использования ожидаемого условия для синхронизации потоков.

27. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Перевод объекта ядра в свободное состояние. Объекты синхронизации: блокировки, семафоры, события.
28. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Ожидаемые таймеры. Оконные таймеры.
29. Структура системного программного интерфейса ОС Windows (Native API). Nt-функции и Zw-функции в пользовательском режиме и режиме ядра ОС Windows.
30. Системный вызов ОС Windows. Алгоритм системного вызова. Особенность системного вызова из режима ядра.
31. Отладка драйверов ОС Windows. Средства отладки драйверов. Посмертный анализ. Живая отладка.
32. Структуры данных общего назначения в режиме ядра ОС Windows. Представление строк стандарта Unicode. Представление двусвязных списков.
33. Механизм прерываний ОС Windows. Аппаратные и программные прерывания. Понятие прерывания, исключения и системного вызова. Таблица векторов прерываний (IDT).
34. Аппаратные прерывания. Программируемый контроллер прерываний. Механизм вызова прерываний. Обработка аппаратных прерываний. Понятие приоритета прерываний (IRQL). Понятие процедуры обработки прерываний (ISR).
35. Понятие приоритета прерываний (IRQL). Приоритеты прерываний для процессора x86 или x64. Процедура обработки прерываний (ISR). Схема обработки аппаратных прерываний.
36. Программные прерывания. Понятие отложенной процедуры (DPC). Назначение отложенных процедур. Механизм обслуживания отложенных процедур. Операции с отложенными процедурами.
37. Понятие асинхронной процедуры (APC). Назначение асинхронных процедур. Типы асинхронных процедур. Операции с асинхронными процедурами.
38. Понятие асинхронной процедуры (APC). Асинхронные процедуры режима ядра: специальная и нормальная APC-процедуры. Асинхронные процедуры пользовательского режима.
39. Понятие элемента работы (Work Item). Назначение элементов работы. Операции с элементами работы. Очереди элементов работы. Обслуживание элементов работы.
40. Управление памятью в ОС Windows. Менеджер памяти. Виртуальная память процесса. Управление памятью в пользовательском режиме. Страничная виртуальная память. Куча (свалка, heap). Проецирование файлов в память.
41. Управление памятью в пользовательском режиме ОС Windows. Оптимизация работы кучи с помощью списков предыстории (Look-aside Lists) и низко-фрагментированной кучи (Low Fragmentation Heap).
42. Структура виртуальной памяти в ОС Windows. Виды страниц. Состояния страниц. Структура виртуального адреса. Трансляция виртуального адреса в физический. Кэширование виртуальных адресов.
43. Управление памятью в режиме ядра ОС Windows. Пулы памяти. Выделение и освобождение памяти в пулах памяти. Структура описателя пула памяти. Доступ к описателям пулов памяти на однопроцессорной и многопроцессорной системах.
44. Пулы памяти ОС Windows. Пул подкачиваемой памяти, пул неподкачиваемой памяти, пул сессии, особый пул. Тегирование пулов. Структура данных пула. Выделение и освобождение памяти в пулах памяти. Организация списков свободных блоков в пуле памяти. Заголовок блока пула памяти.
45. Управление памятью в режиме ядра ОС Windows. Оптимизация использования оперативной памяти с помощью списков предыстории – Look-aside Lists.
46. Представление объекта ядра в памяти. Менеджер объектов.
47. Фиксация данных в физической памяти ОС Windows. Таблица описания памяти (MDL) и ее использование.
48. Понятие драйвера ОС Windows. Виды драйверов. Типы драйверов в режиме ядра. Точки входа в драйвер.
49. Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Структура и взаимосвязь объектов.
50. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при открытии файла.
51. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при выполнении чтения-записи файла.
52. Перехват API-вызовов ОС Windows в пользовательском режиме. Внедрение DLL с помощью реестра. Внедрение DLL с помощью ловушек. Внедрение DLL с помощью дистанционного потока.
53. Перехват API-вызовов ОС Windows в пользовательском режиме. Замена адреса в таблице импорта. Перехват в точке входа в процедуру с помощью подмены начальных инструкций (Microsoft Detours).
54. Перехват API-вызовов ОС Windows в режиме ядра. Таблица системных функций KeServiceDescriptorTable. Таблица системных функций KeServiceDescriptorTableShadow. Понятие UI-потока. Защита от перехвата (Kernel Patch Protection) в 64-разрядной ОС Windows.
55. Перехват API-вызовов менеджера объектов ОС Windows в режиме ядра.
56. Перехват API-вызовов создания и уничтожения процессов и потоков ОС Windows в режиме ядра.
57. Перехват операций с реестром в ОС Windows в режиме ядра.
58. Перехват операций с файлами в ОС Windows в режиме ядра. Мини-фильтры файловой системы.