**1. 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. При этом на экране может быть одновременно выведено несколько 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. 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 является дочерним по отношению к окну Object Window. Это окно не отображается и используется системой Presentation Manager для собственных нужд. Родительское окно может иметь несколько **дочерних окон**. У каждого дочернего окна только одно родительское. Важной особенностью дочерних окон является то, что они всегда располагаются внутри своего родительского окна. Если пользователь попытается переместить дочернее окно за пределы родительского, 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; }

**1.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);

Источники сообщений: Пользователь генерирует сообщения воздействуя на внешние устройства; cама 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 **1.**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.

2.**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.

2. **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. 3)Все элементы графического изображения описываются в рамках логической системы координат[[1]](#footnote-1), которая может отличаться от физической. 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*** 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 );

**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.

**typedef struct tagLOGPALETTE**

**{**

**WORD palVersion;**

**WORD palNumEntries; //** размер палитры (количество эл в массиве структур PALETTEENTRY).

**PALETTEENTRY palPalEntry[1];** // массив структур, описывающих содержимое палитры

**} LOGPALETTE;**

**2. 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. ВAND-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 (16х16, 32х32, 16х32, 64х64).

**Курсоры.** Указатели мыши. Небольшой образ. По своему представлению в файле и памяти курсор напоминает значки, но существуют некоторые значки. Курсоры могут быть размером 16х16 и 32х32. 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.

**2. 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:**

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.

**2. Системы координат.**

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.

DX=(LX-XWO)\*XVE/XWE+XVO DY=(LY-YWO)\*YVE/YWE+YVO

Где: LX – координата Х в логической системе XWO – смещение по оси Х в логической системе

XVO – смещение по оси Х в физической системе координат XVE/XWE – масштабный интерфейс по оси Х

В ОС существуют функции, которые выполняют заданные преобразования для массива точек: **LPtoDP()** и **DPtoLP()**.

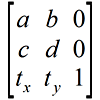
**Матрица трансформации:**

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)

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.

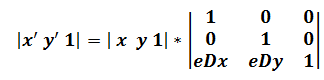
b - Наклон по горизонтали. Положительное значение наклоняет влево, отрицательное вправо.

c - Наклон по вертикали. Положительное значение наклоняет вверх, отрицательное вниз.

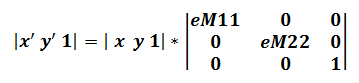
d - Изменение масштаба по вертикали. Значение больше 1 расширяет элемент, меньше 1 — сжимает.

tx - Смещение по горизонтали в пикселах. Положительное значение сдвигает элемент вправо на заданное число пикселов, отрицательное значение сдвигает влево.

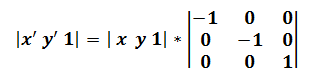
ty - Смещение по вертикали в пикселах. При положительном значении элемент опускается на заданное число пикселов вниз или вверх при отрицательном значении.

**Виды трансформаций**

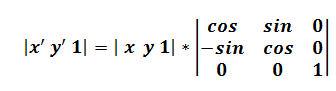
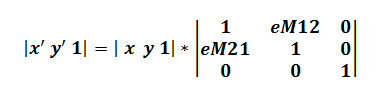
Параллельный перенос:



Масштабирование:



Отражение:



Поворот Наклон:

**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.

**3. 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 автоматически при компиляции DLL. При подключении DLL необходимо знать 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.

**ФУНКЦИЯВХОДА*/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 С++:

\_\_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.

**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.**

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 С++:

\_\_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.

**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](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) или среды программирования обеспечивать возможность параллельной (или [псевдопараллельной](http://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C&action=edit&redlink=1)) 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](http://www.intuit.ru/studies/courses/10471/1078/literature#literature.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

**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](http://msdn.microsoft.com/ru-ru/library/system.threading.threadpool(v=vs.110).aspx) 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](MSDN-PooledThreads.mht)

-[Thread Pooling](MSDN-ThreadPooling.mht)

-[Thread Pool API](MSDN-ThreadPoolAPI.mht)

Класс [ThreadPool](http://msdn.microsoft.com/ru-ru/library/system.threading.threadpool(v=vs.110).aspx) 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**](javascript:void(0))

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](http://msdn.microsoft.com/ru-ru/library/system.threading.threadpool(v=vs.110).aspx) находятся в многопоточном апартаменте.

- Необходимо иметь стабильную идентификацию, сопоставленную с потоком, или назначить поток задаче.

[**Характеристики пула потоков**](javascript:void(0))

-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](http://msdn.microsoft.com/ru-ru/library/system.threading.threadabortexception(v=vs.110).aspx) создается в потоке пула потоков вследствие вызова перегрузки [Abort](http://msdn.microsoft.com/ru-ru/library/system.threading.thread.abort(v=vs.110).aspx).

- Исключение [AppDomainUnloadedException](http://msdn.microsoft.com/ru-ru/library/system.appdomainunloadedexception(v=vs.110).aspx) создается в потоке пула потоков вследствие выгрузки домена приложения.

- Среда CLR или процесс основного приложения прерывает выполнение потока.

[QueueUserWorkItem](http://msdn.microsoft.com/en-us/library/ms684957%28VS.85%29.aspx). 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](http://msdn.microsoft.com/en-us/library/ms685061%28VS.85%29.aspx) 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](http://msdn.microsoft.com/en-us/library/ms682485%28VS.85%29.aspx) — 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-я происх. между 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 сriticalSection; CONDITION\_VARIABLE conditionVariable;

**EnterCriticalSection**(&сriticalSection);

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**(&сriticalSection);

throw; }

**LeaveCriticalSection**(&сriticalSection);

Др. поток заходит в КС

// CRITICAL\_SECTION сriticalSection; CONDITION\_VARIABLE conditionVariable;

**EnterCriticalSection**(&сriticalSection);

try{

ChangeData(); // процедура программиста

**WakeAllConditionVariableCS**(&conditionVariable);}

catch (...){

**LeaveCriticalSection**(&сriticalSection);

throw;}

**LeaveCriticalSection**(&сriticalSection);

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);**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.

**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.

**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, содержащей 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 хранятся указатели на объекты прерываний, каждый из кототрых 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***](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)*(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) – “**RETurn 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.

**На х86: 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:** через 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 х86: 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 х64/IA64 – IRQ number/16

**Программные прерывания – не значит, что через int дёгаем прерывания**

**2 – DISPATCH(dpc)\_LEVEL:** работает планировщик потоков (диспетчер) и отложенные прлоцедуры 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 прерывают весь остальной код

**0 – PASSIVE\_LEVEL(low\_level)**

**НЕ ПУТАТЬ ПРИОРИТЕТЫ ПРЕР-Й И ПРИОРИТЕТЫ ПОТОКОВ (ОНИ НА 0 УРОВНЕ)!!!**

**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) – “**RETurn from Interrupt**“.**

Особенности процедур обработки прерываний:

-- 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).

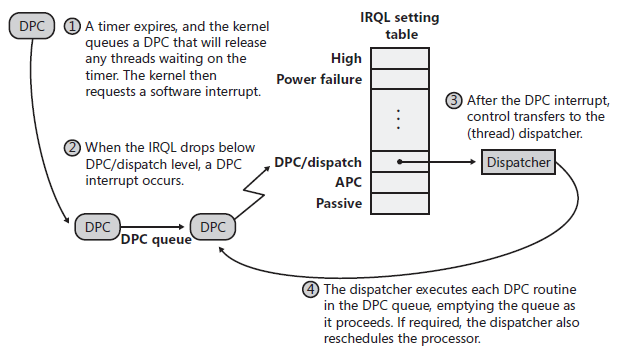
**36. Программные прерывания. Понятие отложенной процедуры (DPC). Назначение отложенных процедур. Механизм обслуживания отложенных процедур. Операции с отложенными процедурами.**

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[].

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**() В пользовательском режиме: **QueueUserApc**()

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.

3**9. 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 IoQueueWorkItemуказывается 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. Так 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. Гранулярность = 2 ссылки, намного меньше, чем в виртуальной памяти – намного лучше.

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 просматривает снова на случай, если подряд идут несколько свободных блоков поменьше – объеденяет в один побольше (функция ядра KiAdjustLookasideDepth() ).

- способ развести потоки и не синхронизировать. Не сработает, только если 2 потока одновременно запросят блок одинакового размера – маловероятно.

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.

- блоки одного размера хранятся на одной странице. Легко дефрагментировать.

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

**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;  
 ……

ULONG TotalBytes;  
 ULONG PoolIndex;  
 LONG TotalPages;  
 VOID\*\* **PendingFrees**;   
 LONG PendingFreeDepth;  
 LIST\_ENTRY **ListHeads**[512]; // 512 двусвязных списков, где лежат блоки, которые можем занимать(были удалены). Гранулярность = 8 байт (размер POOL\_HEADER). Первый(нулевой) список не имеет эл., т.к. всего занимает 8 байт = 8 header + 0 полезных.   
};

**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.

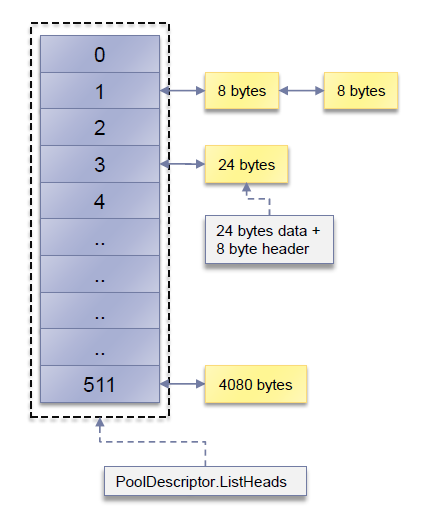
**Выделение памяти в пуле:**

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 0х20 (пробел) до 0х126 (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.

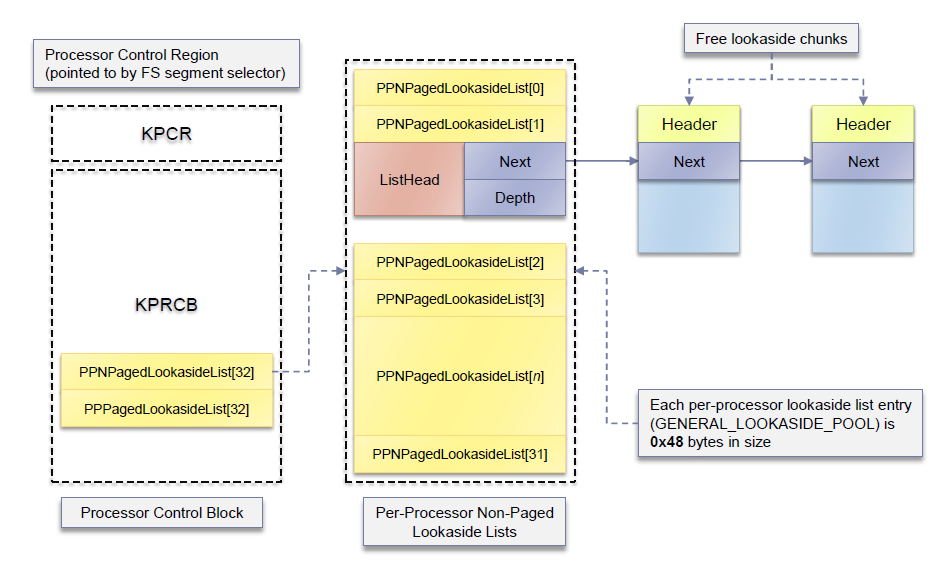
PoolType – тип пула (резидентный, страничный, сеансовый, т.д.).

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.

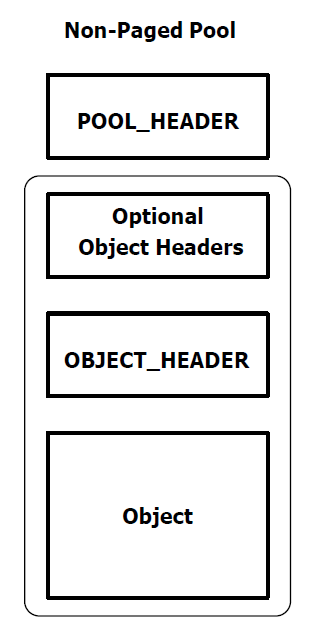
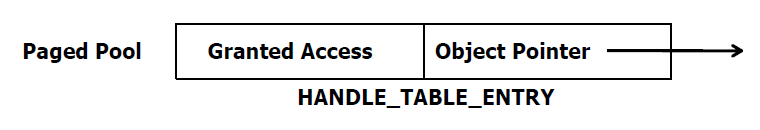
**46. Представление объекта ядра в памяти. Менеджер объектов.**

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.

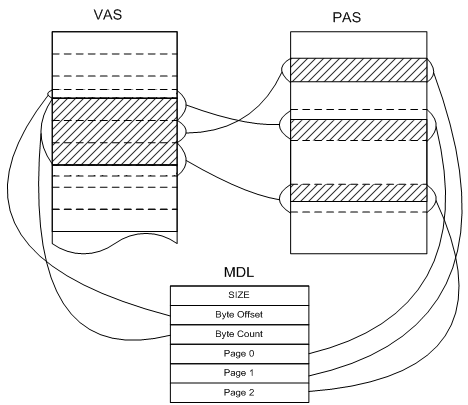
Выделить блок в страничном пуле и зафиксировать его страницы в физической памяти.

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).



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).

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.

**Объект DRIVER\_OBJECT:**

Объект, описывающий драйвер. Соответствует программному модулю драйвера. Содержит список создаваемых драйвером устройств.  
 Хранит состояние, общее для всех обслуживаемых драйвером устройств.

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.

**Объект DEVICE\_OBJECT:**

Контролируемое драйвером физическое или логическое устройство. Содержит указатель на объект, описывающий драйвер. 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.

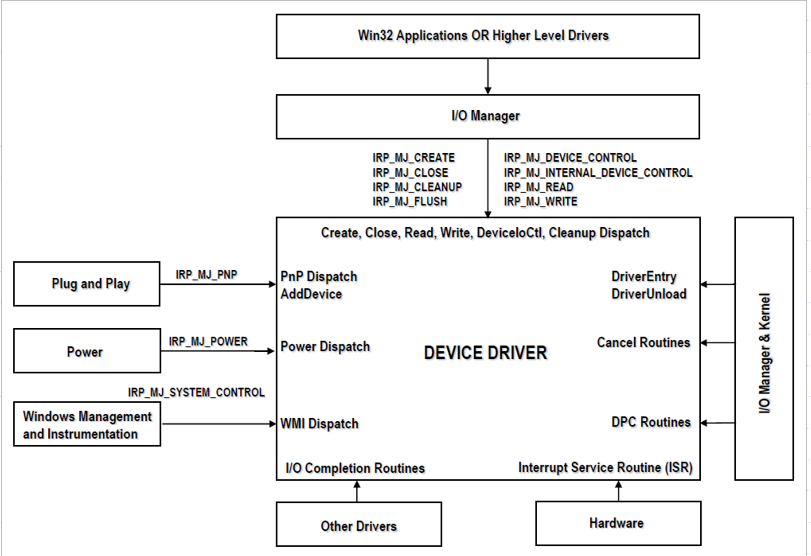
**Объект FILE\_OBJECT:**

Представляет файл, открытый на устройстве. Создается вызовом **CreateFile**()/**ZwCreateFile**().Удаляется вызовом **CloseHandle**()/**ZwClose**(). Открытый на устройстве файл. Содержит указатель на DRIVER\_OBJECT. **Содержит:** Объект устройства – DeviceObject.Относительное имя файла, интерпретируемое драйвером устройства или драйвером файловой системы, – FileName.Дополнительные данные, необходимые драйверам для работы с файлом. Обработка запросов выполняется асинхронно.

**Мой вариант:**

Все драйвера при установке прописываются в реестр, поэтому система знает, у кого вызывать DRIVER\_ENTRY. Когда система видит устройство, она вызывает DRIVER\_ENTRY, которая создаёт DRIVER\_OBJECT. Программист заполняет в ней поля, указатели на процедуры, отдаёт системе (не описывает устройство!)   
Система вызывает ADD\_DEVICE – создаётся DEVICE\_OBJECT по кол-ву физически подключенных устройств.  
  
**Вызов функции:**

Отправка сообщения драйверу с кодом ф-ции, которая нужна, т.е. всё идёт через input-ouput request package.

**Виды отладки в режиме ядра: Посмертный анализ (postmortem analysis), Живая отладка (live debugging).** 

**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](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://ru.wikipedia.org/wiki/Windows%26amp;sa%3DD%26amp;usg%3DAFQjCNF3XHVbBtalFUA3-HEh1mTGNGQMHg&sa=D&ust=1451846696345000&usg=AFQjCNG-2z9KW2LoffBffVDR_wSBJkTevQ), обеспечивающая обмен данными между приложениями и [драйвером](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://ru.wikipedia.org/wiki/%2525D0%252594%2525D1%252580%2525D0%2525B0%2525D0%2525B9%2525D0%2525B2%2525D0%2525B5%2525D1%252580%26amp;sa%3DD%26amp;usg%3DAFQjCNH2PW2JUzbLRYxhbemh-u7AZW6P1A&sa=D&ust=1451846696346000&usg=AFQjCNEIsV7gd_nP2cRnMlX5ScIuHvBkYw), 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.

**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);

**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.

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

Требования к драйверам, применяющим эти механизмы: должен быть скомпонован с ключом /integritycheck, 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.

**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.

1. [↑](#footnote-ref-1)