Операционные системы и системное программирование

Кирилл Сурков, Дмитрий Сурков, Юрий Четырько

Контакты:

kirill.surkov@gmail.com

http://vk.com/kirill.surkov

Литература

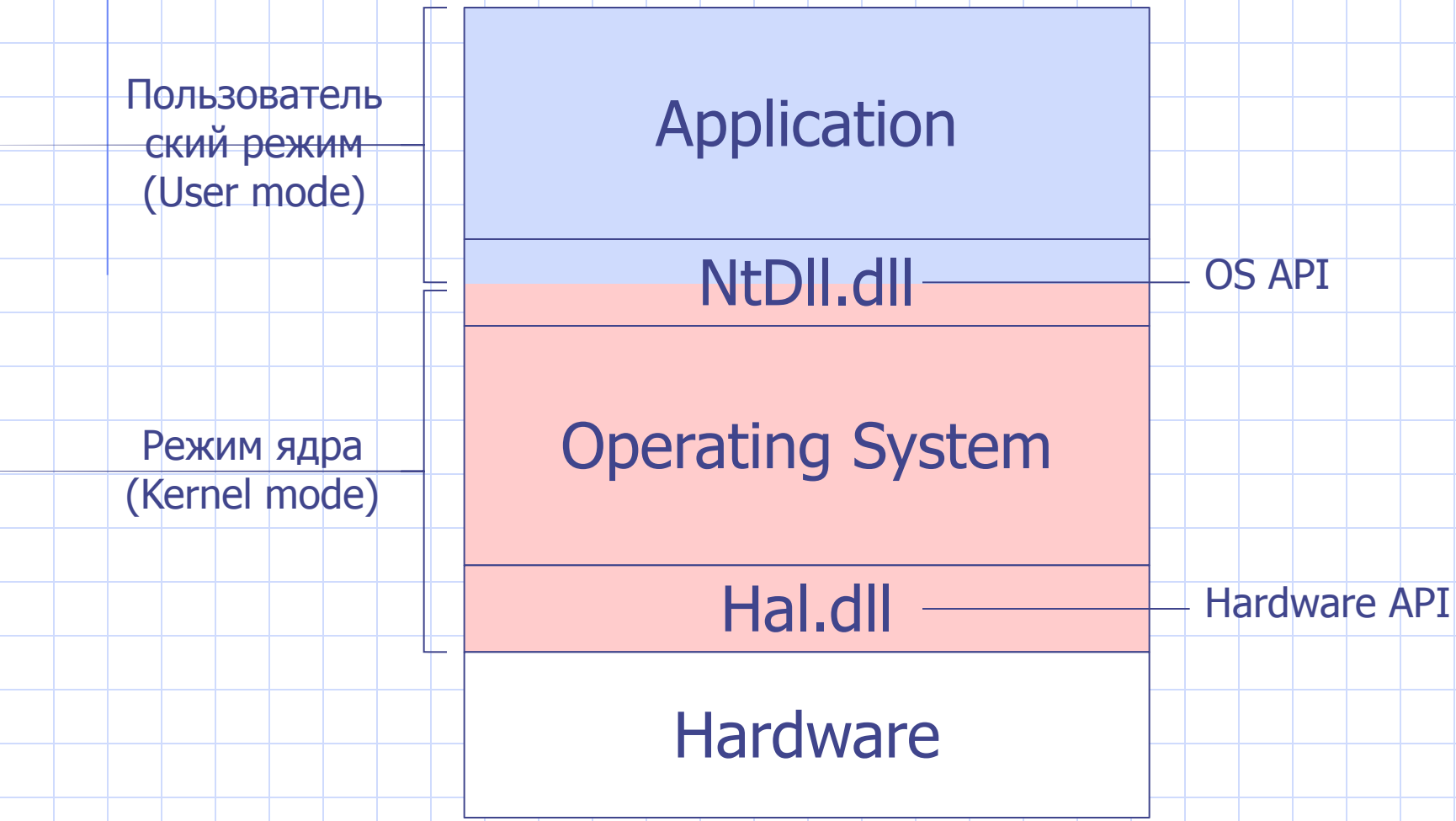
- Данная презентация служит планом изучения предмета и содержит основные темы.
- «Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows». Джеффри Рихтер.
- «Внутреннее устройство Microsoft Windows». Марк Руссинович, Дэвид Соломон.
- «Undocumented Windows NT». Prasad Dabak, Sandeep Phadke, Milind Borate.
- «Windows NT, 2000 Native API Reference». Gary Nebbett.
- «Programming the Microsoft Windows Driver Model». Walter Oney.
- «Developing Drivers with the Microsoft Windows Driver Foundation». Penny Orwick, Guy Smith.
- «Программирование драйверов для Windows». Валерия Комиссарова.
- «Программирование драйверов Windows». Вячеслав Солдатов.
- «Windows 7 Device Driver». Addison Wesley.
- «Современные микропроцессоры». Виктор Корнеев, Андрей Киселев.

Введение в ОС Windows

- Модель программного интерфейса операционной системы Windows.
  - API (Application Programming Interface).
  - Процедурный API. Единая точка доступа к службе – за вызовом процедуры стоит программное прерывание.
  - Объектный подход. Отсутствие указателей на внутренние структуры данных ОС. Применение описателей (дескрипторов) вместо указателей.
  - «Венгерская» нотация в идентификаторах.
- Средства программирования: Visual Studio, Delphi.

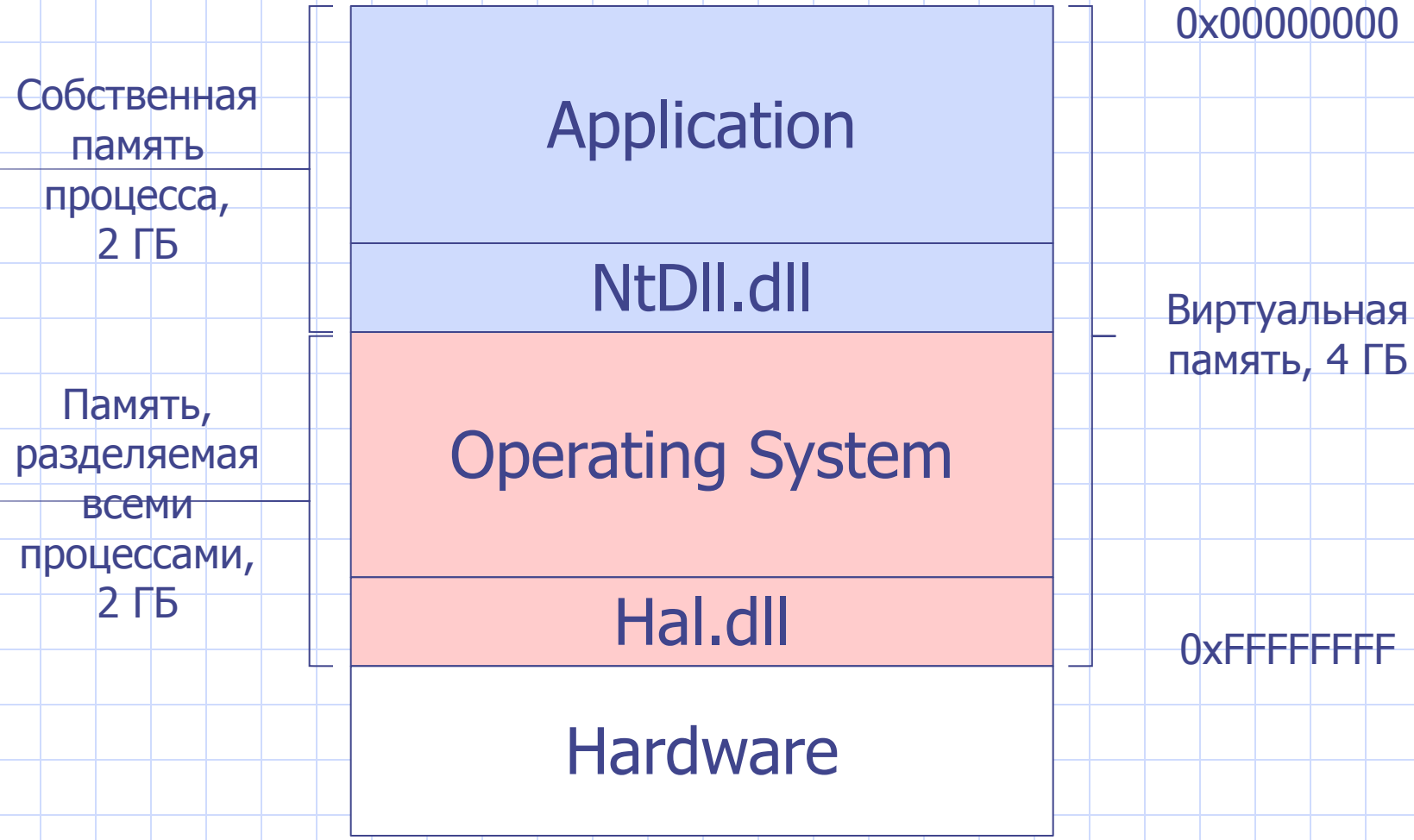
Введение в ОС Windows

- Упрощенная модель архитектуры ОС:



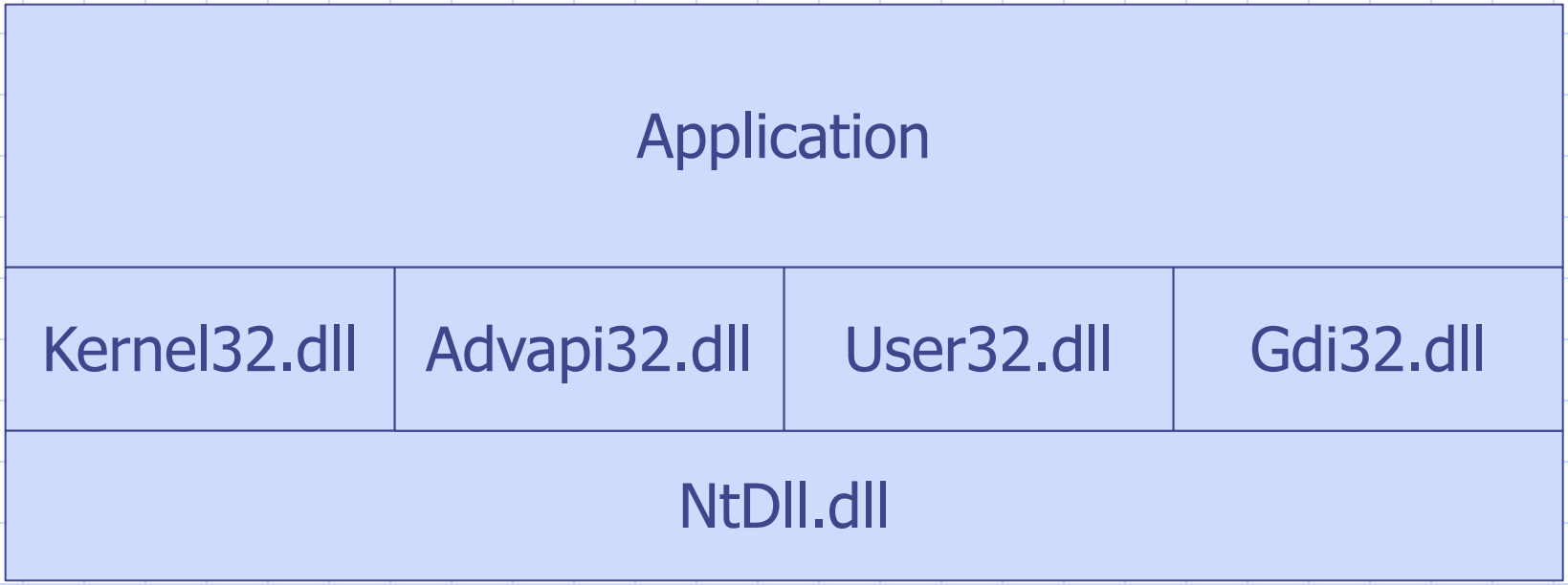
Введение в ОС Windows

- Упрощенная модель памяти (32-разрядной ОС):



Введение в ОС Windows

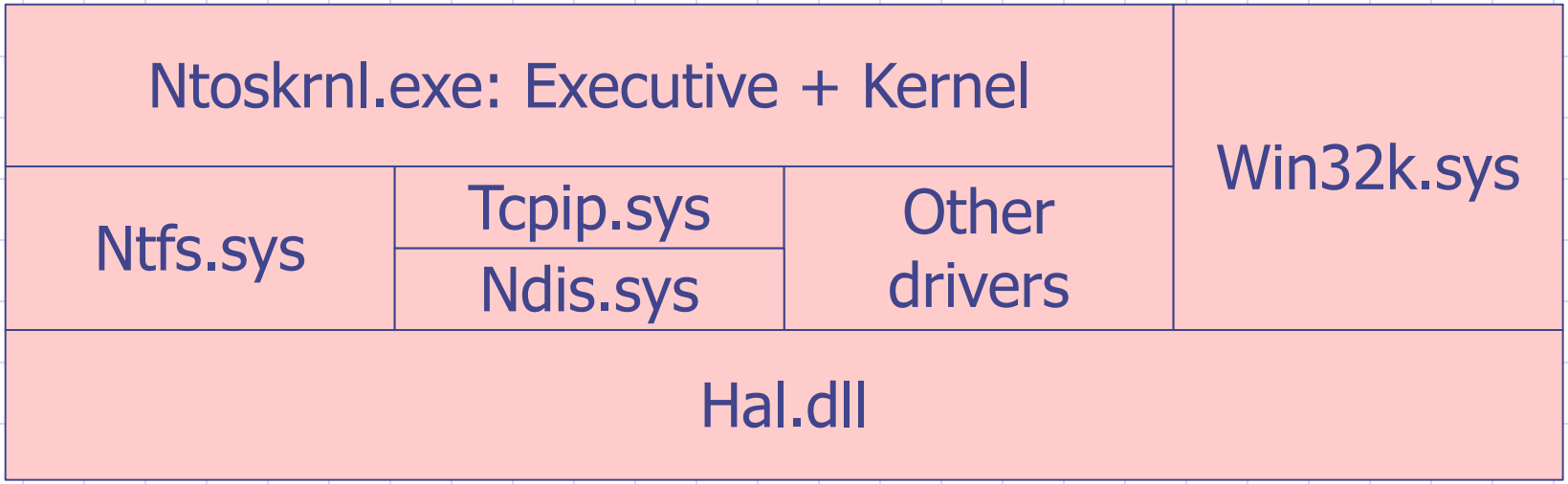
- Архитектура приложения в пользовательском режиме (32-разрядная ОС):



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

Введение в ОС Windows

- Архитектура системы в режиме ядра:



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

Введение в ОС Windows

- Реестр Windows:
  - Иерархическая БД, состоящая из «ульев» – hives.
  - Средства редактирования: regedit.exe, reg.exe. Список средств: http://se-mensh.narod.ru/System/system-reg.htm
  - Главные разделы: HKEY_LOCAL_MACHINE, HKEY_USERS.
  - Короткие пути: HKEY_CURRENT_CONFIG , HKEY_CLASSES_ROOT, HKEY_CURRENT_USER.
  - Точки автозапуска программ при старте ОС.

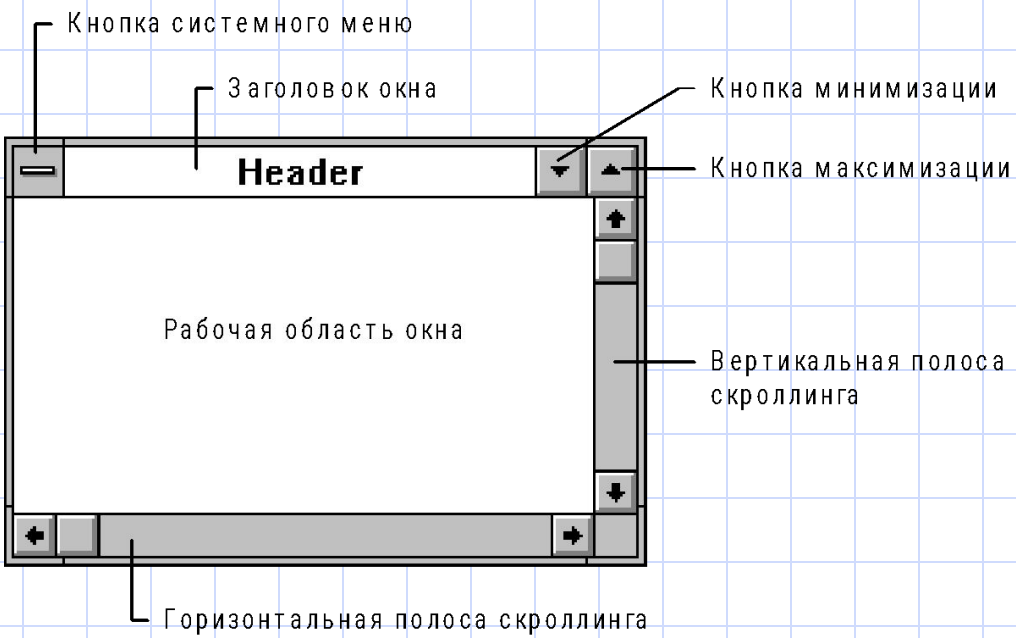
Оконный пользовательский интерфейс

- Нотация Windows API (Win32, Win64):
  - Имена функций – «глагол–существительное»: CreateWindow, ReadFile, SendMessage.
  - Имена переменных – префикс (венгерская нотация, Charles Simonyi).

Префикс	Описание
a	Массив (array)
b	Булевское (boolean)
by	Байт (byte)
c	Счетчик (counter)
ch	Символ (char)
dw	Двойное слово (double word)
fn	Функция (function)
i	Целое (integer)
l	Длинное (long)
n	Целое число (number)
p	Указатель (pointer)
s	Строка (string)
sz	Строка формата ASCIIZ
w	Слово (word)
x	Координата X
y	Координата Y

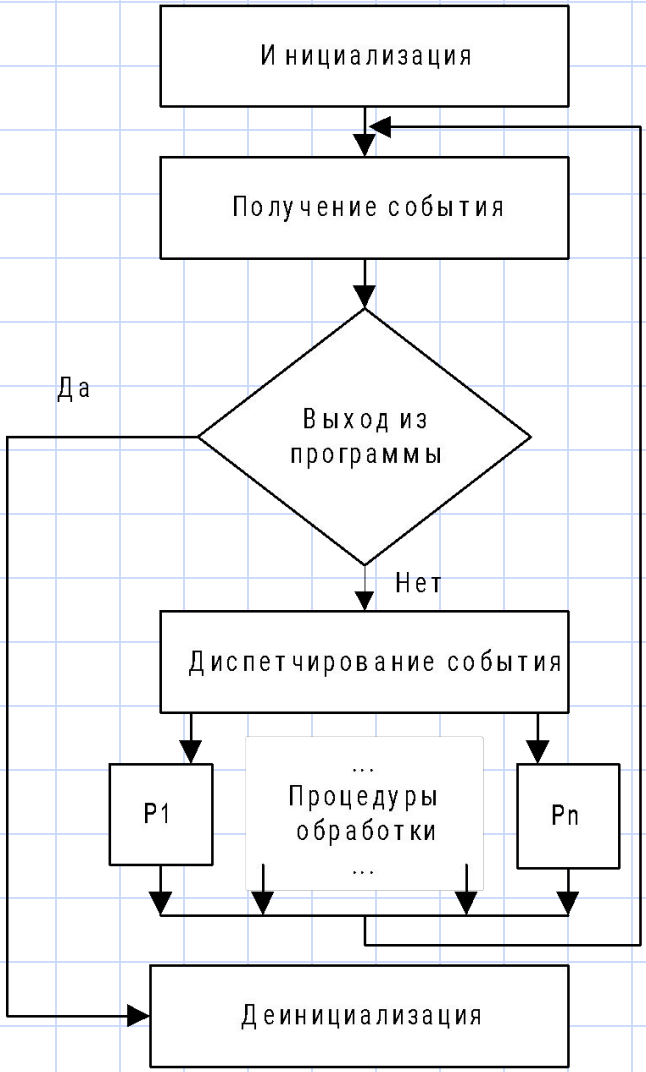
Оконный пользовательский интерфейс

- Элементы окна:



- Понятия родительского и дочернего окна.

Программа с событийным управлением



Сообщение Windows

```
struct MSG
{
    HWND hwnd;      // Описатель окна, в котором возникло
                    // сообщение.

    UINT message;   // Код сообщения: WM_<сообщение>,
                    // пользовательские начинаются с WM_USER.

    WPARAM wParam; // Доп. информация (зависит от сообщения)


    LPARAM lParam; // Доп. Информация (зависит от сообщения)


    DWORD time;     // Время в миллисекундах с момента
                    // запуска системы до постановки
                    // сообщения в очередь.
    POINT pt;       // Позиция курсора мыши в экранных
                    // координатах на момент возникновения
                    // сообщения.
};
```

Минимальная программа для Windows

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInstance,
     HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
     WNDCLASSEX wcex; HWND hWnd; MSG msg;

     wcex.cbSize = sizeof(WNDCLASSEX);
     wcex.style = CS_DBLCLKS;
     wcex.lpfnWndProc = WndProc;
     wcex.cbClsExtra = 0;
     wcex.cbWndExtra = 0;
     wcex.hInstance = hInstance;
     wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
     wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
     wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
     wcex.lpszMenuName = NULL;
     wcex.lpszClassName = "HelloWorldClass";
     wcex.hIconSm = wcex.hIcon;

...
```

Минимальная программа для Windows

```
...

    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;
}

...
```

Минимальная программа для Windows

```
...

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;
}
```

Посылка сообщений

```
VOID PostQuitMessage(int nExitCode);

BOOL PostThreadMessage(DWORD idThread, UINT Msg,
    WPARAM wParam, LPARAM lParam);

BOOL PostMessage(HWND hWnd, UINT Msg,
    WPARAM wParam, LPARAM lParam);

LRESULT SendMessage(HWND hWnd, UINT Msg,
    WPARAM wParam, LPARAM lParam);
```

Обработка сообщений

- Обработка сообщений мыши:
  - WM_LBUTTONDOWN, WM_LBUTTONUP, WM_MOUSEMOVE
  - CS_DBLCLKS – WM_LBUTTONDBLCLK
  - wParam – состояние кнопок мыши и клавиш Ctrl & Shift.
  - lParam – младшие 2 байта кодируют координату X, старшие – координату Y.

Обработка сообщений

- Обработка сообщений клавиатуры:
  - WM_KEYDOWN, WM_KEYUP
  - WM_SYSKEYDOWN, WM_SYSKEYUP – с клавишей Alt.
  - wParam – виртуальный код клавиши VK_X.
  - lParam – счетчик повтора, индикатор расширенной клавиши, системной клавиши (удерживался ли Alt), предыдущего состояния (была ли до этого нажата клавиша), индикатор текущего состояния.
  - TranslateMessage – WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, WM_SYSDEADCHAR.
  - short GetKeyState(int vKey), GetAsyncKeyState(int vKey) – состояние клавиш Ctrl и Shift.
  - TranslateAccelerators – WM_COMMAND

Вывод информации в окно

- Перерисовка окна:
  - Понятие области обновления – UPDATE REGION, WM_PAINT.
  - bool InvalidateRect(HWND, const RECT*, bool bErase); bErase – посылать WM_ERASEBKGND перед WM_PAINT. Парная процедура – ValidateRect.
  - bool InvalidateRgn(HWND, HRGN, bool bErase); Парная процедура – ValidateRgn.
  - WM_NCPAINT, WM_PAINT.
  - void UpdateWindow(HWND); – немедленный вызов WM_NCPAINT и WM_PAINT.

Вывод информации в окно

- Обработка сообщения WM_PAINT:
  - HDC BeginPaint(HWND, PAINTSTRUCT*);
    Посылает WM_ERASEBKGND, если необходимо.
  - Рисование: DC – Device Context.
    bool Ellipse(HDC, int, int, int, int);
    bool Rectangle(HDC, int, int, int, int);
    …
  - bool EndPaint(HWND, const PAINTSTRUCT*);
- Рисование в любой момент времени:
  - HDC GetDC(HWND);
  - int ReleaseDC(HWND, HDC);
  - CS_OWNDC – флаг оконного класса.

Графическая система Windows

- Управление цветом:
  - typedef DWORD COLORREF;
  - COLORREF color = RGB(255, 0, 0); // 0x000000FF
  - BYTE GetRValue(DWORD rgb), GetGValue, GetBValue.
  - COLOREF GetNearestColor(HDC, COLORREF);
  - COLORREF SetBkColor(HDC, COLORREF), GetBkColor.
  - LOGPALETTE, CreatePalette, SetPaletteEntries, GetPaletteEntries, SelectPalette, RealizePalette, DeleteObject.

Графическая система Windows

- Инструменты для рисования:
  - DC – 1 Bitmap , 1 Region, 1 Pen, 1 Brush, 1 Palette, 1 Font.
  - HPEN – LOGPEN, CreatePenIndirect, CreatePen.
  - HBRUSH – LOGBRUSH, CreateBrushIndirect, CreateBrush.
  - HFONT – LOGFONT, CreateFontIndirect, CreateFont.
  - HANDLE GetStockObject(int);
  - HANDLE SelectObject(HDC, HANDLE);
  - bool DeleteObject(HANDLE);

Растровые изображения

- Виды растровых изображений:
  - Bitmap – базовый формат растрового изображения.
  - Icon – значок: AND-маска и XOR-маска.
  - Cursor – курсор: две маски и точка касания – Hot Spot.
- 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.
  - XOR-маска – цветная. Фигура кодируется цветом, прозрачный фон – нулем. Врезает на экране фигуру на месте «черной дыры».

Растровые изображения Bitmap

- Два типа растровых изображений Bitmap:
  - Device-Dependent Bitmap – аппаратно-зависимое. Точки находятся в прямом соответствии с пикселями экрана или другой поверхности отображения. Быстрый формат, но не универсален.
  - Device-Independent Bitmap – аппаратно-независимое. 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. Медленный, но универсальный формат. Применяется для хранения растрового изображения на диске. Возможно применение алгоритма сжатия Run Length Encoding (RLE).

Работа с растровыми изображениями

- Загрузка, копирование, вывод на устройство:
  - HBITMAP **LoadBitmap**(HINSTANCE hInstance, LPCTSTR lpBitmapName), **LoadIcon**, **LoadCursor**.
  - HANDLE **LoadImage**(HINSTANCE hinst, LPCTSTR lpszName, UINT uType, int cxDesired, int cyDesired, UINT fuLoad);
  - HANDLE **CopyImage**(HANDLE hImage, UINT uType, int cxDesired, int cyDesired, UINT fuFlags);
  - BOOL **DrawIcon**(HDC hDC, int X, int Y, HICON hIcon), **DrawIconEx** – рисование и значка, и курсора.
  - Вывести Bitmap вызовом одной функции нельзя. Необходимо создать дополнительное устройство в памяти – memory DC, выбрать в нем Bitmap в качестве поверхности рисования, выполнить перенос изображения из memory DC в window DC.

Вывод растрового изображения

```
void ShowBitmap(HWND hWnd, HBITMAP hBmp)
{
    HDC winDC = GetDC(hWnd);
    HDC memDC = CreateCompatibleDC(winDC);
    HBITMAP oldBmp = SelectObject(memDC, hBmp);
    BitBlt(winDC, 10, 10, 64, 64, memDC, 0, 0, SRCCOPY);
    SelectObject(memDC, oldBmp);
    DeleteDC(memDC);
    ReleaseDC(hWnd, winDC);
}
```

Вывод растрового изображения

- bool **BitBlt**(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight, HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop);
- bool **StretchBlt**(HDC hdcDest, int nXOriginDest, int nYOriginDest, int nWidthDest, int nHeightDest, HDC hdcSrc, int nXOriginSrc, int nYOriginSrc, int nWidthSrc, int nHeightSrc, DWORD dwRop);
- bool **AlphaBlend**(HDC hdcDest, int xoriginDest, int yoriginDest, int wDest, int hDest, HDC hdcSrc, int xoriginSrc, int yoriginSrc, int wSrc, int hSrc, BLENDFUNCTION ftn);
- int **StretchDIBits**(HDC hdc, int XDest, int YDest, int nDestWidth, int nDestHeight, int XSrc, int YSrc, int nSrcWidth, int nSrcHeight, const VOID *lpBits, const BITMAPINFO *lpBitsInfo, UINT iUsage, DWORD dwRop);

Библиотека Direct2D

- Загрузка, копирование, вывод на устройство:
  - Direct2D QuickStart
  - Direct2D Application

Вывод текста

- bool **TextOut**(HDC hdc, int nXStart, int nYStart, LPCTSTR lpString, int cchString); **SetTextAlign**, **GetTextAlign**, **SetTextColor**, **GetTextColor**, **SetTextJustification**, **GetTextJustification**, **SetTextCharacterExtra**, **GetTextCharacterExtra**.
- bool **ExtTextOut**(HDC hdc, int X, int Y, UINT fuOptions, const RECT* lprc, LPCTSTR lpString, UINT cbCount, const int* lpDx); **GetTextExtentPoint**, **GetTextExtentExPoint**.
- bool **PolyTextOut**(HDC hdc, const POLYTEXT* pptxt, int cStrings);
- long **TabbedTextOut**(HDC hDC, int X, int Y, LPCTSTR lpString, int nCount, int nTabPositionCount, const int* lpnTabPositions, int nTabOrigin); **GetTabbedTextExtent**.
- int **DrawText**(HDC hDC, LPCTSTR lpchText, int nCount, RECT* lpRect, UINT uFormat);
- int **DrawTextEx**(HDC hdc, LPTSTR lpchText, int cchText, RECT* lprc, UINT dwDTFormat, DRAWTEXTPARAMS* lpDTParams);

Шрифты

- Шрифт – множество символов со сходными размерами и начертанием контуров. Параметры:
  - Гарнитура (typeface) – с засечками, без засечек.
  - Начертание (style) – полужирный, курсив.
  - Кегль (size) – размер в пунктах, 10 pt = 3.76 мм.
- Семейство шрифта – набор шрифтов со сходной шириной символов и гарнитурой:
  - Decorative – декоративный (Wingdings)
  - Modern – моноширинный (Courier New)
  - Roman – с засечками (Times New Roman)
  - Swiss – без засечек (Arial)
  - Script – рукописный (Script MT Bold)

Шрифты

- Тип шрифта:
  - Растровый
  - Векторный
  - TrueType – сплайны 3-го порядка;
  - OpenType – сплайны 2-го (Adobe PostScript) или 3-го порядка.
- Шрифты в программе:
  - Физические – устанавливаемые в операционную систему, файлы.
  - Логические – запрашиваемые программой у операционной системы, LOGFONT.

Физический шрифт

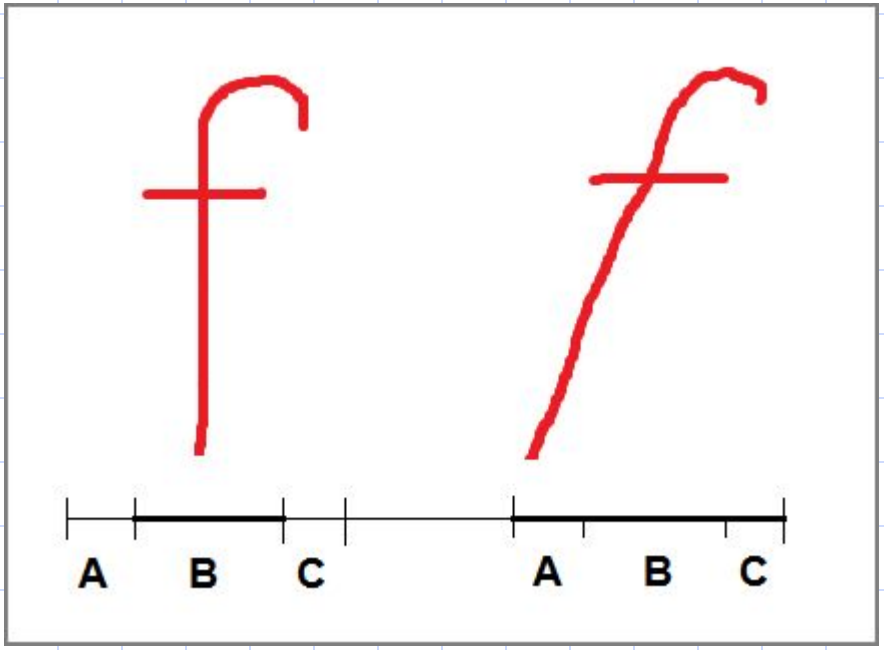
- Установка шрифта:
  - Скопировать файл шрифта в C:\Windows\Fonts.
  - Вызвать int AddFontResource(LPCTSTR lpszFilename).
  - Вызвать SendMessage с кодом WM_FONTCHANGE.
- Удаление шрифта:
  - Вызвать bool RemoveFontResource(LPCTSTR lpszFilename).
  - Удалить файл шрифта из C:\Windows\Fonts.
  - Вызвать SendMessage с кодом WM_FONTCHANGE.
- Временная установка шрифта:
  - Не копировать файл в C:\Windows\Fonts, или
  - AddFontMemResourceEx и RemoveFontMemResourceEx.

Логический шрифт

- Создание логического шрифта (LOGFONT):
  - CreateFontIndirect / CreateFont,
  - SelectObject,
  - DeleteObject.
- Поиск логического шрифта:
  - int **EnumFonts**(HDC hdc, LPCTSTR lpFaceName, FONTENUMPROC lpFontFunc, LPARAM lParam); перечисление через вызов callback-процедуры.
  - **EnumFontFamilies**, **EnumFontFamiliesEx**.

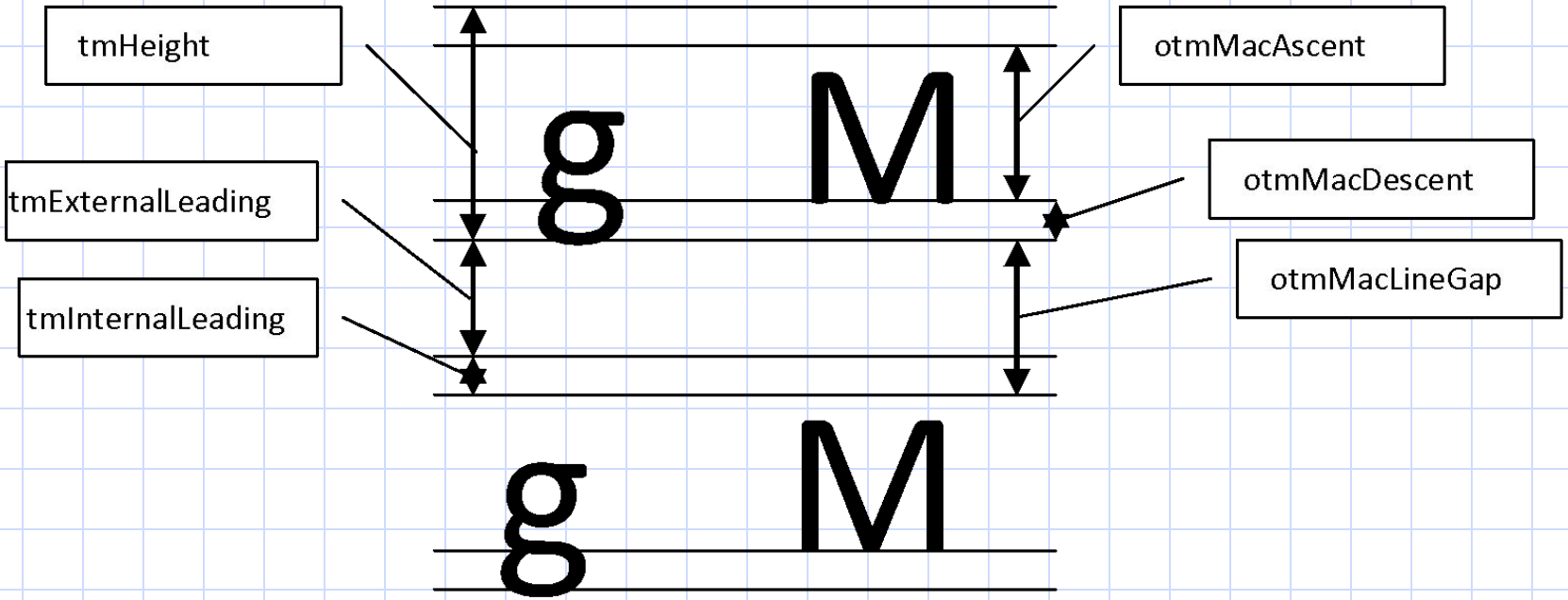
Логический шрифт

- Параметры ширины:
  - bool **GetCharABCWidths**(HDC hdc, UINT uFirstChar, UINT uLastChar, ABC* lpabc);
  - **GetCharABCWidthsFloat**, **GetCharABCWidthsI**.



Логический шрифт

- Параметры высоты:
  - bool **GetTextMetrics**(HDC hdc, **TEXTMETRIC*** lptm);
  - UINT **GetOutlineTextMetrics**(HDC hdc, UINT cbData, **OUTLINETEXTMETRIC*** lpOTM) – для шрифта TrueType.



Системы координат

- 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
- Матрица трансформации:

```
struct XFORM
{
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
};
```

```
| eM11  eM12  0 |
| eM21  eM22  0 |
| eDx   eDy   1 |
```

Трансформации

- Применение матрицы трансформации:

```
                                                | eM11  eM12  0 |
| x′   y′   1 |   =   | x   y   1 |   *        | eM21  eM22  0 |
                                                | eDx   eDy   1 |
```

- Формулы:
  - x' = x * eM11 + y * eM21 + eDx
  - y' = x * eM12 + y * eM22 + eDy
- Функции:
  - bool SetWorldTransform(HDC hdc, const XFORM* lpXform);
  - bool ModifyWorldTransform(HDC hdc, const XFORM* lpXform, DWORD iMode);
  - bool GetWorldTransform(HDC hdc, XFORM* lpXform);

Трансформации

- Параллельный перенос:

```
                                    |  1    0    0 |
| x′  y′  1 |   =   | x   y   1 | * |  0    1    0 |
                                    | eDx  eDy   1 |
```

- Масштабирование:

```
                                    | eM11   0    0 |
| x′  y′  1 |   =   | x   y   1 | * |  0   eM22   0 |
                                    |  0     0    1 |
```

- Отражение:

```
                                    | -1    0    0 |
| x′  y′  1 |   =   | x   y   1 | * |  0   -1    0 |
                                    |  0    0    1 |
```

Трансформации

- Поворот:

```
                                        | cos   sin   0 |
| x′  y′  1 |   =   | x   y   1 |  *   |-sin   cos   0 |
                                        |  0     0    1 |
```

- Наклон:

```
                                        |  1   eM12   0 |
| x′  y′  1 |   =   | x   y   1 |  *   | eM21  1     0 |
                                        |  0     0    1 |
```

Страничная система координат

- Преобразования в страничной системе координат:
  - DX = (LX – XWO) * XVE/XWE + XVO
  - DY = (LY – YWO) * YVE/YWE + YVO
  - LX, LY – координаты в логической системе
  - DX, DY – координаты в физической системе
- Функции:
  - bool LPtoDP(HDC hdc, POINT* lpPoints, int nCount), DPtoLP.
  - SetWindowOrgEx, SetViewportOrgEx – смещения.
  - SetViewportExtEx, SetWindowExtEx – масштабные коэффициенты. Взятые отдельно, эти функции смысла не имеют.

Страничная система координат

- Режимы масштабирования:
  - int SetMapMode(HDC hdc, int fnMapMode), GetMapMode.

Режим масштабирования	Логических единиц	Физических единиц	Направление осей	
			X	Y
MM_TEXT(default)	1	1 pixel	→	↓
MM_LOMETRIC	10	1 mm	→	↑
MM_HIMETRIC	100	1 mm	→	↑
MM_LOENGLISH	100	1 inch	→	↑
MM_HIENGLISH	1000	1 inch	→	↑
MM_TWIPS	1440	1 inch	→	↑
MM_ISOTROPIC	Задается	Задается	→	↑
MM_ANISOTROPIC	Задается	Задается	→	↑

Ресурсы прикладной программы

- Ресурсы – двоичные данные, записываемые в исполняемый модуль приложения. Стандартные виды ресурсов:
  - Курсор – Cursor
  - Картинка – Bitmap
  - Значок – Icon
  - Меню – Menu
  - Окно диалога – Dialog Box
  - Таблица строк – String Table
  - Таблица сообщений (об ошибках) – Message Table
  - Шрифт – Font
  - Таблица горячих клавиш – Accelerator Table
  - Информация о версии – Version Information
  - Ресурс Plug and Play
  - Ресурс VXD
  - Ресурс HTML
  - Манифест приложения – Side-by-Side Assembly Manifest
  - Двоичные данные – RCData

Ресурсы прикладной программы

- Добавление и удаление ресурсов исполняемого модуля:
  - HANDLE **BeginUpdateResource**(LPCTSTR pFileName, bool bDeleteExistingResources);
  - bool **UpdateResource**(HANDLE hUpdate, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage, void* lpData, DWORD cbData);
  - bool **EndUpdateResource**(HANDLE hUpdate, bool fDiscard);
- Загрузка ресурсов из исполняемого модуля:
  - HRSRC **FindResourceEx**(HMODULE hModule, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage); **FindResource**, **EnumResourceXxx**.
  - HGLOBAL **LoadResource**(HMODULE hModule, HRSRC hResInfo); **LoadImage**, **LoadMenu**, **LoadXxx**.
  - DWORD **SizeofResource**(HMODULE hModule, HRSRC hResInfo);

Динамически-загружаемые библиотеки

- Динамически-загружаемая библиотека (DLL) – двоичный модуль операционной системы. Это программа с множеством точек входа. Включает код, данные и ресурсы.

- Подключение DLL называется импортом. Существуют статический импорт и динамический импорт. При статическом импорте динамическая библиотека подключается как статическая, но находится в отдельном исполняемом файле и поэтому может быть заменена перед стартом. При динамическом импорте загрузка и получение адресов функций динамической библиотеки происходит вручную во время работы программы.

Статический импорт DLL-библиотеки

- Экспорт функции при создании DLL:
  - __declspec(dllexport) int Min(int X, int Y);
- Импорт функции из DLL:
  - Добавить библиотеку DLL в проект Visual Studio.
  - __declspec(dllimport) int Min(int X, int Y);
  - При сборке проекта будет создана статическая библиотека импорта с расширением LIB. Эта статическая библиотека включается в EXE файл и содержит код вызова функции Min из DLL-библиотеки.

Статический импорт DLL-библиотеки

- Соглашения о вызовах подпрограмм:
  - __declspec(dllimport) int __stdcall Min(int X, int Y);
  - **__cdecl** – Параметры передаются на стек в обратном порядке. За освобождение стека после вызова под-программы отвечает вызывающая программа.
  - **__pascal** – Передача параметров на стек в прямом порядке. Освобождение стека осуществляет сама вызванная подпрограмма.
  - **__stdcall** – Соглашение для стандартных DLL ОС Windows. Передача параметров на стек происходит в обратном порядке. Освобождение стека выполняет вызванная подпрограмма.
  - **__register** – Передача параметров преимущественно через регистры процессора. Не используется при создании DLL, поскольку не стандартизировано.

Статический импорт DLL-библиотеки

- Соглашения о вызовах подпрограмм:
  - Разные способы передачи параметров создают трудности. Главная трудность связана с применением соглашения __stdcall. В VisualStudio использование соглашение о вызовах __stdcall вводит определенные правила именования функций в DLL. Функция получает имя: _Имя@КоличествоБайтПараметров.

    _Min@8

  - Библиотека импорта может создаваться вручную на основе существующей DLL библиотеки. Для этого создается текстовый DEF-файл описания DLL библиотеки и включается в проект.

    EXPORTS
    Min
    Max

  - При наличии DEF-файла компилятор выбирает из него имена для функций.

Динамический импорт DLL-библиотеки

- 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**.

Динамический импорт DLL-библиотеки

- 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);

Точка входа-выхода

- Точка входа-выхода – функция DllMain (имя не закреплено):
  - bool WINAPI **DllMain**(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved);
  - Можно вызывать лишь функции Kernel32.dll. Нельзя вызывать функции LoadLibrary, LoadLibraryEx и функции других DLL.
- Сценарии:
  - **DLL_PROCESS_ATTACH** – первая загрузка DLL каким-либо потоком.
  - **DLL_THREAD_ATTACH** – подключение нового потока. Для первого потока не применяется.
  - **DLL_THREAD_DETACH** – упорядоченное завершение потока (например, с помощью ExitThread).
  - **DLL_PROCESS_DETACH** – упорядоченное завершение процесса (например, с помощью ExitProcess).
  - **Осторожно!** Снятие процесса (TerminateProcess) или потока (TerminateThread) не приводит к вызову функции DllMain.

Динамически-загружаемые библиотеки

- Разделяемые данные – shared data:
  - #pragma section("mysection", read, write, shared)
    __declspec(allocate("mysection")) int Number = 0;
- Переадресация к процедуре в другой DLL:
  - #pragma comment(linker, "/export:MyProc=OtherDll.OtherProc")
  - В разделе экспорта DLL для процедуры MyProc создается переадресация к процедуре OtherProc в OtherDll.
  - Просмотр раздела экспорта:
    C:\>dumpbin -exports MyDll.dll

Динамически-загружаемые библиотеки

- Исключение конфликта версий DLL:
  - **c:\myapp\myapp.exe** загружает старую версию **c:\program files\common files\system\mydll.dll**, а должен загружать **mydll.dll** из своего же каталога.
  - Создать пустой файл **c:\myapp\myapp.exe.local**. Будет грузиться библиотека **c:\myapp\mydll.dll**.
  - Создать каталог **c:\myapp\myapp.exe.local**. Будет грузиться **c:\myapp\myapp.exe.local\mydll.dll**.
  - Создать файл манифеста для приложения. В этом случае .local файлы будут игнорироваться.

Объекты ядра

- Виды объектов:
  - Объекты оконной системы – User Objects
  - Объекты графической системы – GDI Objects
  - Объекты ядра – Kernel Objects
- Объекты ядра с атрибутами защиты:
  - Access Token
  - Communications device
  - Console input
  - Console screen buffer
  - Desktop
  - Directory
  - Event
  - File
  - File mapping
  - Job
  - Mailslot
  - Mutex
  - Pipe
  - Process
  - Registry key
  - Semaphore
  - Socket
  - Thread
  - Timer
  - Window station

Объекты ядра

- Атрибуты защиты – **SECURITY_ATTRIBUTES**:
  - struct SECURITY_ATTRIBUTES
    {
    DWORD nLength;
    void* lpSecurityDescriptor;
    bool bInheritHandle;
    };
- Дескриптор защиты – **SECURITY_DESCRIPTOR**:
  - Owner security identifier (SID)
  - Primary group SID
  - Discretionary access control list (DACL)
  - System access control list (SACL)
- Функции создания и редактирования дескриптора защиты:
  - bool **InitializeSecurityDescriptor**(PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD dwRevision);
  - **SetSecurityDescriptorXxx**, **GetSecurityDescriptorXxx**.

Объекты ядра

- Создание объекта ядра:
  - HANDLE **CreateXxx**(SECURITY_ATTRIBUTES* lpAttributes, ..., LPCTSTR lpName);
  - HANDLE **OpenXxx**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
  - bool **DuplicateHandle**(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, HANDLE hTargetProcessHandle, HANDLE* lpTargetHandle, DWORD dwDesiredAccess, bool bInheritHandle, DWORD dwOptions);
  - bool **SetHandleInformation**(HANDLE hObject, DWORD dwMask, DWORD dwFlags); HANDLE_FLAG_INHERIT, HANDLE_FLAG_PROTECT_FROM_CLOSE.
  - bool **GetHandleInformation**(HANDLE hObject, DWORD* lpdwFlags);
- Удаление объекта ядра:
  - bool **CloseHandle**(HANDLE hObject);

Проецирование файлов в память

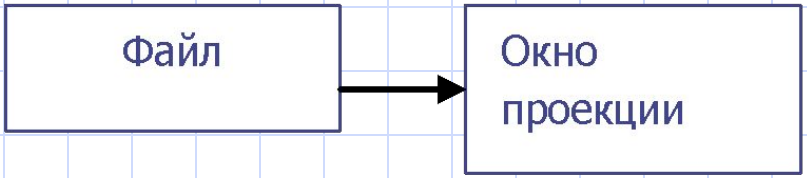
- Постановка задачи:
  - Закрепить за началом файла какой-либо адрес памяти и выполнять чтение и запись файла методом чтения и записи байтов оперативной памяти.
- Реализация:
  - Поскольку файл не может поместиться в оперативной памяти целиком, он делится на страницы и в оперативную память подгружаются лишь те страницы, к которым происходит обращение. Адресное пространство файла является виртуальным, оно может значительно превосходить по размерам оперативную память. Для прозрачной поддержки проецирования файлов в память необходимо иметь поддержку виртуальной памяти на уровне процессора и архитектуры компьютера.
  - Для процессов ОС Windows, работающих в виртуальном адресном пространстве, на диске создается файл подкачки (pagefile.sys). При проецировании файлов в память, файл подкачки не затрагивается. Это обеспечивается тем, что в таблице страниц виртуальной памяти для каждой страницы сохраняется ссылка на файл, из которого загружается страница.

Проецирование файлов в память

- Отличие между Windows и UNIX/Linux:

В ОС Win отображение файлов в память является двухуровневым:

В UNIX схема проецирования файлов одноуровневая:



Проецирование файлов в память

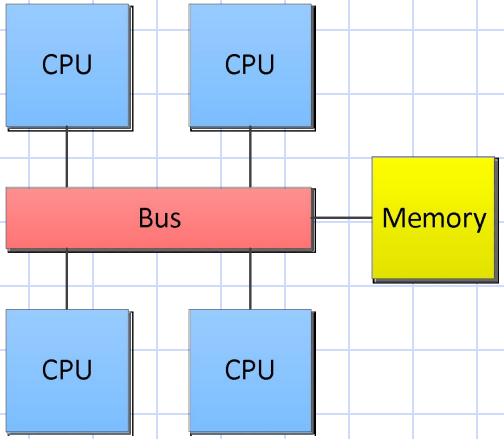
- Создать объект ядра – файл:
  - HANDLE **CreateFile**(LPCTSTR lpFileName,
    DWORD dwDesiredAccess, DWORD dwShareMode,
    SECURITY_ATTRIBUTES* lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
- Создать объект ядра – проекция файла:
  - HANDLE **CreateFileMapping**(HANDLE hFile,
    SECURITY_ATTRIBUTES* lpAttributes, DWORD flProtect,
    DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow,
    LPCTSTR lpName); **CreateFileMappingNuma**.
- Создать окно проекции:
  - void* **MapViewOfFileEx** (HANDLE hFileMappingObject,
    DWORD dwDesiredAccess, DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap,
    void* lpBaseAddress); **MapViewOfFile**.

Проецирование файлов в память

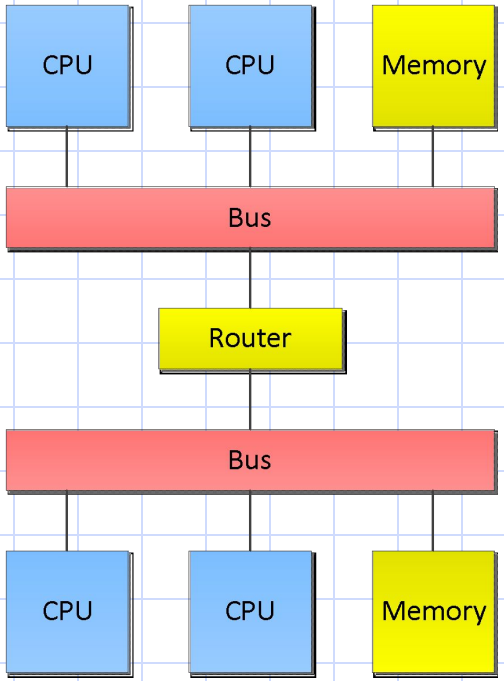
- Синхронизировать память с диском:
  - bool **FlushViewOfFile** (const void* lpBaseAddress, SIZE_T dwNumberOfBytesToFlush);
- Закрыть окно проекции:
  - bool **UnmapViewOfFile**(const void* lpBaseAddress);
- Закрыть файл:
  - bool **CloseHandle**(HANDLE hObject);

Современные многопроцессорные архитектуры

- Симметричная многопроцессорная архитектура – **SMP** (Symmetric Multi-Processor Architecture)

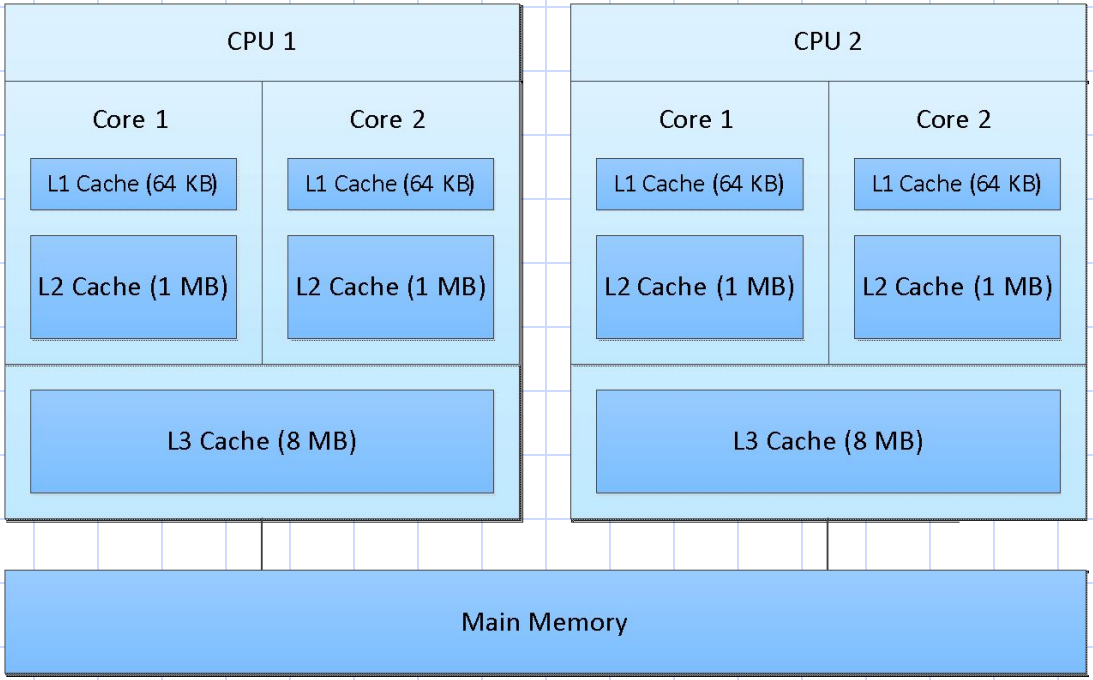


- Гибридная архитектура с неоднородным доступом к памяти – **NUMA** (Non-Uniform Memory Access Architecture)



Современные многопроцессорные архитектуры

- Кэширование памяти
  - Уровни кэш-памяти – **L1**, **L2**, **L3**
  - Кэш-память разных процессоров может быть когерентной и некогерентной;
  - Если кэш-память не является когерентной, в многопоточных приложениях может происходить перестановка операций чтения и записи.



Современные многопроцессорные архитектуры

- Перестановка операций чтения и записи
  - Из-за оптимизаций компилятора;
  - Из-за наличия кэша в процессоре.
- Рассмотрим пример – переменная модифицируется внутри блокировки (критической секции), затем блокировка снимается:

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

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

Современные многопроцессорные архитектуры

- Решение проблемы перестановки операций чтения и записи
  - Синхронизация кэшей – **cache coherence**;
  - Барьеры памяти – **memory barrier** (**memory fence**).
- Устранение проблемы с помощью барьера памяти:

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

- В процессорах x86 и SPARC применяется строгая модель памяти (**strong memory ordering**), а именно, модель со строгим порядком записи – **total store ordering** (**TSO**):
  - Чтения упорядочиваются в соответствии с предыдущими чтениями;
  - Записи упорядочиваются в соответствии с предыдущими чтениями и записями;
  - Это означает, что чтения могут «проглядеть» предыдущие записи, но не могут проглядеть предыдущие чтения, а записи не могут «проглядеть» предыдущие чтения и записи.

Средства распараллеливания

- Процесс (process)
  - Процесс – выполняемая программа, имеющая собственное виртуальное адресное пространство, код, данные, а также потребляющие ресурсы ОС.
  - Надежное средство. Аварийное завершение процесса не приводит к утечке ресурсов или нарушению целостности данных в других процессах.
  - Менее эффективное средство. Поскольку процессы работают в разных адресных пространствах, необходимо использовать средства Inter-Process Communication (IPC) для доступа к общим данным.
- Поток/Нить (thread)
  - Поток – выполняемая подпрограмма процесса, разделяющая с другими потоками общие ресурсы процесса.
  - Эффективное средство. 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.
  - Менее надежное средство. 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.

Процессы

- Создание:
  - 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);
  - **CreateProcessAsUser**, **LogonUser**, **CreateProcessWithLogonW**, **CreateProcessWithTokenW**, **ShellExecute**.
- Поиск выполняемого файла:
  - Каталог EXE-файла вызывающего процесса.
  - Текущий каталог вызываемого процесса.
  - Системный каталог Windows.
  - Основной каталог Windows.
  - Каталоги, перечисленные в переменной окружения PATH.

Процессы

- Дополнительные начальные параметры – **STARTUPINFO**:
  - struct STARTUPINFO
    {
      DWORD cb; LPTSTR lpReserved; LPTSTR lpDesktop; LPTSTR lpTitle;
      DWORD dwX; DWORD dwY; DWORD dwXSize; DWORD dwYSize;
      DWORD dwXCountChars; DWORD dwYCountChars;
      DWORD dwFillAttribute; DWORD dwFlags; WORD wShowWindow;
      WORD cbReserved2; LPBYTE lpReserved2;
      HANDLE hStdInput; HANDLE hStdOutput; HANDLE hStdError;
    };
- Информация о процессе – **PROCESS_INFORMATION**:
  - struct PROCESS_INFORMATION
    {
      HANDLE hProcess; //описатель процесса
      HANDLE hThread; //описатель потока в пределах текущего процесса
      DWORD dwProcessId; //уникальный id процесса в пределах системы
      DWORD dwThreadId; //уникальный id потока в пределах системы
    };

Процессы

- Полезные функции:
  - HANDLE **GetCurrentProcess**(void); // CloseHandle ничего не делает
  - HANDLE **GetCurrentThread**(void); // CloseHandle ничего не делает
  - DWORD **GetCurrentProcessId**(void);
  - DWORD **GetCurrentThreadId**(void);
  - void **ExitProcess**(UINT uExitCode); // код возврата всех потоков
  - bool **TerminateProcess**(HANDLE hProcess, UINT exitCode);
  - bool **GetExitCodeProcess**(HANDLE hProcess, DWORD* exitCode);
  - DWORD **WaitForInputIdle**(HANDLE hProcess, DWORD millisec);
- Запуск процесса по цепочке:

```
CreateProcess(…, &pi); // PROCESS_INFORMATION pi;
CloseHandle(pi.hThread);
WaitForSingleObject(pi.hProcess);
GetExitCodeProcess(pi.hProcess, &exitCode); // DWORD exitCode;
CloseHandle(pi.hProcess);
```

Потоки

- Создание и завершение:
  - HANDLE **CreateThread**(
    SECURITY_ATTRIBUTES* lpThreadAttributes, SIZE_T dwStackSize,
    THREAD_START_ROUTINE* lpStartAddress, void* lpParameter,
    DWORD dwCreationFlags, DWORD* lpThreadId);
  - DWORD WINAPI **ThreadProc**(void* lpParameter);
  - dwCreationFlags: CREATE_SUSPENDED
  - HANDLE **CreateRemoteThread**(HANDLE hProcess,
    SECURITY_ATTRIBUTES* lpThreadAttributes, SIZE_T dwStackSize,
    THREAD_START_ROUTINE* lpStartAddress, void* lpParameter,
    DWORD dwCreationFlags, DWORD* lpThreadId);
  - **CreateRemoteThreadEx** – дополнительный параметр PROC_THREAD_ATTRIBUTE_LIST* lpAttributeList. Можно задать процессор, на котором запустится поток.
  - void **ExitThread**(DWORD dwExitCode);
  - bool **TerminateThread**(HANDLE hThread, DWORD dwExitCode);

Потоки

- Приостановка и возобновление потока (не применять):
  - DWORD **SuspendThread**(HANDLE hThread);
  - DWORD **ResumeThread**(HANDLE hThread);
- Контекст потока – запись, сохраняющая состояние потока:
  - struct _CONTEXT // специфична для процессора x86

```
{
    DWORD ContextFlags;
    DWORD Dr0; DWORD Dr1; DWORD Dr2;
    DWORD Dr3; DWORD Dr6; DWORD Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD SegGs; DWORD SegFs; DWORD SegEs; DWORD SegDs;
    DWORD Edi; DWORD Esi; DWORD Ebx; DWORD Edx;
    DWORD Ecx; DWORD Eax; DWORD Ebp;
    DWORD Eip; DWORD SegCs; DWORD EFlags;
    DWORD Esp; DWORD SegSs;
    BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

Пул потоков (thread pool)

- Почему нужен пул потоков?
  - Старт нового потока занимает много времени.
  - Количество процессоров ограничено.
- Архитектура пула потоков:
  - Рабочие потоки (worker threads) вызывают callback-функции.
  - Ожидающие потоки ждут на объектах ожидания (wait handles).
  - Очередь рабочих потоков (work queue).
  - Стандартный пул потоков на каждый процесс.
  - Менеджер рабочих потоков (worker factory).
  - Стандартный размер пула потоков – 500 рабочих потоков.
  - Pooled Threads: Improve Scalability With New Thread Pool APIs
  - Thread Pooling
  - Thread Pool API

Распределение процессорного времени

- Понятие уровня приоритета:
  - 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.
  - Уровни приоритета для потоков присваиваются в 2 этапа:
    1. Процессу присваивается класс приоритета.
    2. Потоку присваивается относительный уровень приоритета.
    Результирующий приоритет определяется как сумма этих двух значений (на самом деле результат определяется по таблице).

Распределение процессорного времени

- Классы приоритета:
  - IDLE_PRIORITY_CLASS 4
  - BELOW_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS 8
  - ABOVE_NORMAL_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS 13
  - REALTIME_PRIORITY_CLASS 24
- Относительные уровни приоритета:
  - THREAD_PRIORITY_IDLE 1 // общий результат
  - THREAD_PRIORITY_LOWEST –2
  - THREAD_PRIORITY_BELOW_NORMAL –1
  - THREAD_PRIORITY_NORMAL +0
  - THREAD_PRIORITY_ABOVE_NORMAL +1
  - THREAD_PRIORITY_HIGHEST +2
  - THREAD_PRIORITY_TIME_CRITICAL 15 // общий результат

Распределение процессорного времени

- 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.

Распределение процессорного времени

- Функции:
  - bool **SetPriorityClass**(HANDLE hProcess, DWORD dwPriorityClass);
  - DWORD **GetPriorityClass**(HANDLE hProcess);
  - bool **SetThreadPriority**(HANDLE hThread, int nPriority);
  - int **GetThreadPriority**(HANDLE hThread);
  - 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);
  - DWORD **SleepEx**(DWORD dwMilliseconds, bool bAlertable);

Механизмы 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

- Критическая секция – небольшой участок кода, требующий монопольного доступа к каким-то общим данным.
  - struct CRITICAL_SECTION
    {
        LONG LockCount;
        LONG RecursionCount;
        HANDLE OwningThread;
        HANDLE LockSemaphore;
        ULONG_PTR SpinCount;
    };
  - void **InitializeCriticalSection**(CRITICAL_SECTION* lpCriticalSection);
  - void **EnterCriticalSection**(CRITICAL_SECTION* lpCriticalSection);
  - void **LeaveCriticalSection**(CRITICAL_SECTION* lpCriticalSection);
  - bool **TryEnterCriticalSection**(CRITICAL_SECTION* lpCriticalSection);
  - bool **InitializeCriticalSectionAndSpinCount**(CRITICAL_SECTION* lpCriticalSection, DWORD dwSpinCount); **SetCriticalSectionSpinCount**.

Ожидаемое условие

- Ожидаемое условие – механизм синхронизации, позволяющий потокам дождаться выполнения некоторого (сложного) условия. Состоит из критической секции и переменной условия.
  - void **InitializeConditionVariable**(CONDITION_VARIABLE* CondVariable);
  - bool **SleepConditionVariableCS**(CONDITION_VARIABLE* CondVariable, CRITICAL_SECTION* CriticalSection, DWORD dwMilliseconds);
  - bool **SleepConditionVariableSRW**(CONDITION_VARIABLE* CondVariable, SRWLOCK* SRWLock, DWORD dwMilliseconds, ULONG Flags);
  - void **WakeConditionVariable**(CONDITION_VARIABLE* CondVariable);
  - void **WakeAllConditionVariable**(CONDITION_VARIABLE* CondVariable);
  - Using Condition Variables

Ожидаемое условие – пример (страница 1)

- Пример использования ожидаемого условия:

```
// CRITICAL_SECTION criticalSection;
// CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try
{
    while (DataDoesntSatisfyCondition()) // функция программиста
        SleepConditionVariableCS(&conditionVariable, &criticalSection, INFINITE);
}
catch (...)
{
    LeaveCriticalSection(&criticalSection);
    throw;
}
LeaveCriticalSection(&criticalSection);
```

Ожидаемое условие – пример (страница 2)

- Пример использования ожидаемого условия:

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

Атомарные операции

- 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**.
  - LONG **InterlockedCompareExchange**(LONG* Destination, LONG Exchange, LONG Comparand); **InterlockedCompareExchangePointer**.
  - **InterlockedBitTestAnd**(**Set**/**Reset**/**Complement**).
  - **InterlockedXxx64**, **InterlockedXxxNoFence**, **InterlockedXxxAcquire**, **InterlockedXxxRelease**.
  - Acquire and Release Semantics
  - Interlocked Singly Linked Lists

Ожидание

- Объекты ядра Windows могут находится в одном из двух состояний:
  - Свободном состоянии (signaled)
  - Занятом (not signaled)
- Синхронизация – ожидание освобождения объекта ядра:
  - DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds);
  - 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**.
  - Wait Functions

Блокировка

- Блокировка – mutex (**mut**ually **ex**clusive), бинарный семафор. Используется для обеспечения монопольного доступа к некоторому ресурсу со стороны нескольких потоков (различных процессов).
  - HANDLE **CreateMutex**(SECURITY_ATTRIBUTES* lpMutexAttributes, bool bInitialOwner, LPCTSTR lpName);
  - HANDLE **OpenMutex**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
  - DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds);
  - bool **ReleaseMutex**(HANDLE hMutex);
  - bool **CloseHandle**(HANDLE hObject);

Семафор

- 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.
  - HANDLE **CreateSemaphore**(SECURITY_ATTRIBUTES* lpSecurityAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);
  - HANDLE **OpenSemaphore**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
  - DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds);
  - bool **ReleaseSemaphore**(HANDLE hSemaphore, LONG lReleaseCount, LONG* lpPreviousCount);
  - bool **CloseHandle**(HANDLE hObject);

Событие

- 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.
- Пример:
  - Выполнение некоторым поток действий в контексте другого потока.
- Функции:
  - 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.
  - bool **CloseHandle**(HANDLE hObject);

Ожидаемый таймер

- Ожидаемый таймер – объект ядра, самостоятельно переходящий в свободное состояние в определенное время и/или через определенные промежутки времени.
  - HANDLE **CreateWaitableTimer**(SECURITY_ATTRIBUTES* lpSecurityAttributes, BOOL bManualReset, LPCTSTR lpTimerName);
  - HANDLE **OpenWaitableTimer**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpTimerName);
  - bool **SetWaitableTimer**(HANDLE hTimer, const LARGE_INTEGER* pDueTime, LONG lPeriod, TIMERAPCROUTINE* pfnCompletionRoutine, void* lpArgToCompletionRoutine, bool fResume);
  - void CALLBACK **TimerAPCProc**(void* lpArgToCompletionRoutine, DWORD dwTimerLowValue, DWORD dwTimerHighValue); – создается программистом; вызывается системой с помощью **QueueUserAPC**.
  - bool **CancelWaitableTimer**(HANDLE hTimer);
  - bool **CloseHandle**(HANDLE hObject);

Оконный таймер

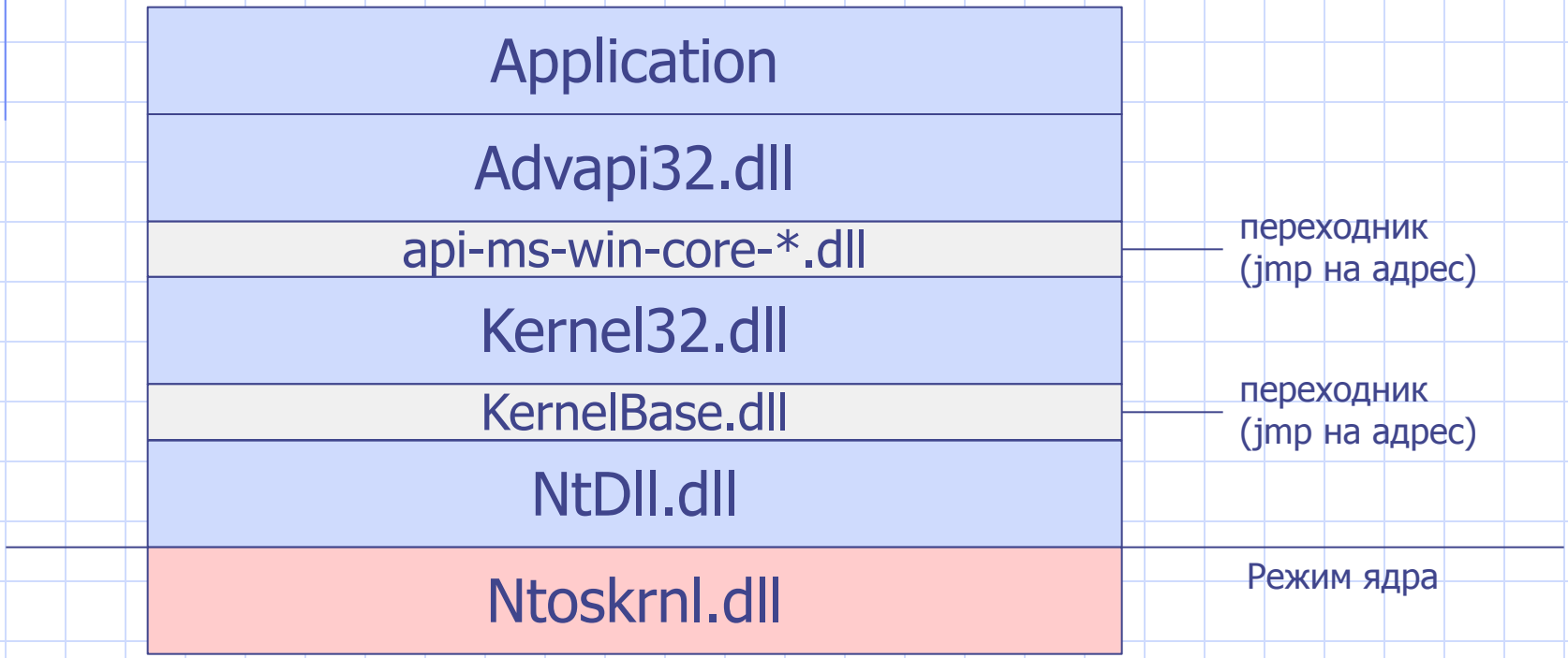
- Оконный таймер – механизм посылки таймерных сообщений через определенные промежутки времени.
  - UINT_PTR **SetTimer**(HWND hWnd, UINT_PTR nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);
  - void CALLBACK **TimerProc**(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD dwTime);
  - bool **KillTimer**(HWND hWnd, UINT_PTR uIDEvent);

Подготовка к системному программированию

- Visual Studio Professional 2013 (или Ultimate-версия):
  - http://msdn.microsoft.com/en-US/windows/hardware/gg454513
- Windows Software Development Kit – SDK 8.1:
  - http://msdn.microsoft.com/en-US/windows/desktop/bg162891
- Windows Driver Kit – WDK 8.1:
  - http://msdn.microsoft.com/en-US/windows/hardware/gg454513
- Символьная информация о системных модулях Windows:
  - http://msdn.microsoft.com/en-US/windows/hardware/gg454513
- Утилита Dependency Walker для просмотра зависимостей DLL-библиотек:
  - http://www.dependencywalker.com/
- Набор утилит Sysinternals для глубокого просмотра процессов, дисков, сети и прочего:
  - http://technet.microsoft.com/en-us/sysinternals/bb545027

Структура системного API – Native API

- Структура системного вызова Windows 7, 8 и выше:
  - http://www.nirsoft.net/articles/windows_7_kernel_architecture_changes.html
  - http://msdn.microsoft.com/en-us/library/windows/desktop/hh802935(v=vs.85).aspx



Структура системного API – Native API

- Все функции ядра Windows, доступные пользовательским приложениям, экспортируются библиотекой NtDll.dll. Системные функции называются Native API. Системные функции имеют префикс Nt, например NtReadFile.
- В режиме ядра дополнительно существуют парные функции с префиксом Zw, например ZwReadFile. Они вызываются драйверами вместо Nt-функций. В пользовательском режиме Zw-имена тоже объявлены, но эквивалентны Nt-именам.
- Каждой Nt-функции сопоставлен номер. Номера зависят от версии Windows, полагаться на них не следует.
- Номер функции – это индекс в двух системных таблицах: nt!KiServiceTable и nt!KiArgumentTable. В первой таблице (System Service Descriptor Table – SSDT) хранятся адреса Nt-функций, во второй таблице – объемы параметров в байтах.
- Глобальная переменная nt!KeServiceDescriptorTable хранит три значения: указатель на таблицу nt!KiServiceTable, указатель на таблицу nt!KiArgumentTable и количество элементов в этих таблицах.

Системный вызов

- Алгоритм системного вызова (действия, выполняемые Nt-функцией библиотеки NtDll.dll):
  - Загрузить в регистр **EAX** номер Nt-функции.
  - Загрузить в регистр **EDX** указатель на вершину параметров в стеке (ESP).
  - Вызвать прерывание для перехода процессора в режим ядра:
    int 0x2E – на старых процессорах,
    sysenter – на современных процессорах Intel,
    syscall – на современных процессорах AMD.
  - Если используется прерывание, 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**().
  - Если используется специальная инструкция (sysenter или syscall), то происходит вызов функции, адрес которой хранится в специальном внутреннем регистре процессора (Model Specific Register – MSR). Этот регистр хранит адрес функции ядра **KiFastCallEntry**().
  - После перехода в режим ядра все параметры, передаваемые в Nt-функцию, находятся на стеке пользовательского режима.

Системный вызов

- Алгоритм системного вызова (продолжение в режиме ядра):
  - По номеру функции в регистре **EAX** отыскать в **nt!KiArgumentTable** количество байтов, занимаемое параметрами на стеке.
  - Скопировать параметры со стека пользовательского режима на стек ядра. После переключение процессора в режим ядра стек тоже переключен.
  - По номеру функции в регистре **EAX** отыскать в **nt!KiServiceTable** адрес функции для вызова.
  - Выполнить вызов функции. Функция выполняется в контексте вызывающего процесса и потока и поэтому обращается к указателям пользовательского режима напрямую.
  - Если функция вызвана из пользовательского режима, выполняется проверка параметров. Скалярные значения проверяются на допустимые диапазоны. Указатели проверяются с помощью функций ProbeForRead() и ProbeForWrite() в блоке __try { } __except { }.
  - Вернуться из режима ядра в пользовательский режим с помощью:
    iret – на старых процессорах,
    sysexit – на современных процессорах Intel,
    sysret – на современных процессорах AMD.

Системный вызов внутри ядра

- Если Nt-функция вызывается внутри ядра:
  - Проверка параметров не выполняется.
  - Такая функция может быть недоступна в ядре, т.е. может не экспортироваться модулем Ntoskrnl.exe.
  - Вызов Nt-функции с передачей ей указателей на память ядра закончится ошибкой.
- Вместо Nt-функций модули ядра вызывают Zw-функции. Zw-функция делает следующее:
  - Загружает в регистр EAX номер функции.
  - Загружает в регистр EDX указатель на вершину параметров в стеке ядра.
  - Вызывает соответствующую ей Nt-функцию. При этом проверка параметров не выполняется.

Отладка драйверов Windows

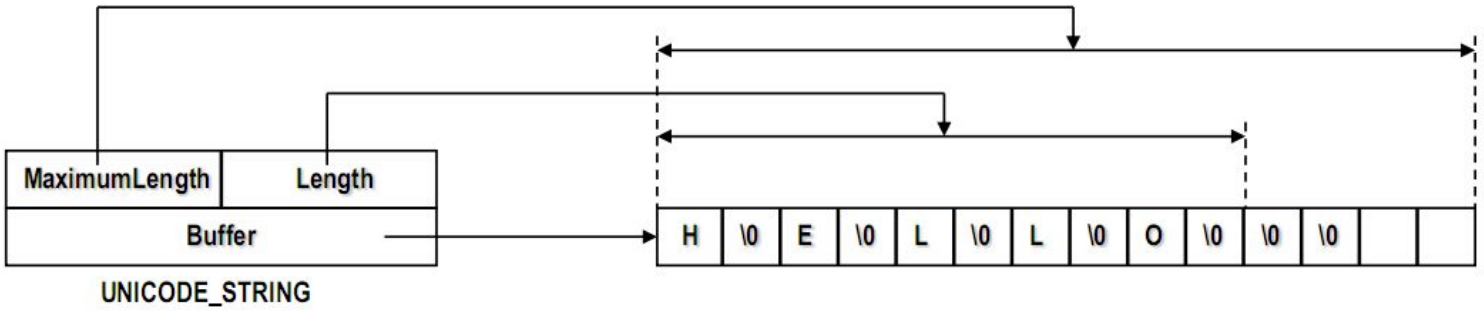
- Средства отладки драйверов:
  - Поддержка отладки ядра обеспечивается самим ядром Windows. Включается: bcdedit /debug on
  - В процессорах архитектуры x86 имеются специальные отладочные регистры DR0-DR7. Они позволяют отладчику ставить контрольные точки на чтение и запись памяти, а также на порты ввода-вывода.
  - Традиционный отладчик с пользовательским интерфейсом – windbg.exe.
  - Отладчик командной строки – kd.exe.
  - Современное средство отладки: Visual Studio 2013 Professional + WDK 8.1.
  - Начиная с Windows Vista, обеспечивается создание ряда драйверов, работающих в пользовательском режиме. Для отладки применяется Visual Studio.

Виды отладки в режиме ядра Windows

- Посмертный анализ (postmortem analysis):
  - Включить в операционной системе создание дампов памяти: Start->Control Panel->System->Advanced system settings->Advanced tab ->Startup and Recovery->Settings->Write debugging information: Small memory dump (256 KB) или Kernel memory dump.
  - Включить отладку в ядре: bcdedit /debug on
  - Настроить канал связи отладчика с ядром: bcdedit /dbgsettings
  - В отладчике включить загрузку символьной информации о ядре Windows: Tools->Options->Debugging->Symbols->[x] Microsoft Symbol Servers.
  - Открыть файл C:\Windows\MEMORY.DMP в отладчике.
- Живая отладка (live debugging):
  - Соединить два компьютера через один из следующих интерфейсов: Serial, IEEE 1394, USB 2.0.
  - Включить отладку в ядре: bcdedit /debug on
  - Настроить канал связи отладчика с ядром: bcdedit /dbgsettings
  - В отладчике включить загрузку символьной информации о ядре Windows: Tools->Options->Debugging->Symbols->[x] Microsoft Symbol Servers.
  - В Visual Studio собрать драйвер.
  - Поставить контрольную точку в исходном коде и установить драйвер.

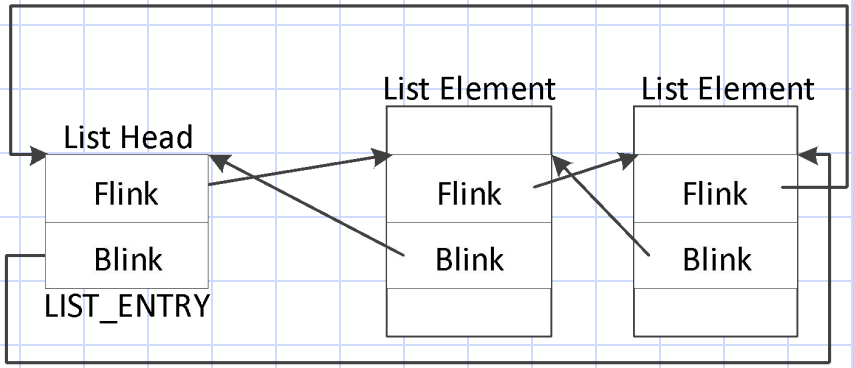
Структуры данных ядра: строки Unicode

- UNICODE_STRING:
  - Ядро Windows хранит строки в формате Unicode. Строки, передаваемые функциям ядра, находятся почти всегда в формате Unicode.
  - struct UNICODE_STRING
    {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
    };
  - Буфер, на который указывает поле Buffer, обычно выделяется из пула подкачиваемой страничной памяти.
  - Поле Length содержит число байтов (не WCHARS). Завершающий символ UNICODE_NULL в это число не включен.



Структуры данных ядра: двусвязные списки

- LIST_ENTRY:
  - Большинство структур данных ядра хранятся как части двусвязных списков.
  - struct LIST_ENTRY
    {
    LIST_ENTRY* Flink;
    LIST_ENTRY* Blink;
    };
  - Поля Flink и Blink в структуре LIST_ENTRY указывают на вложенную структуру LIST_ENTRY, а не на начало элемента списка.
  - Структуры LIST_ENTRY никогда не содержат нулевых указателей. Когда список пуст, поля Flink и Blink в голове списка ListHead указывают непосредственно на ListHead.

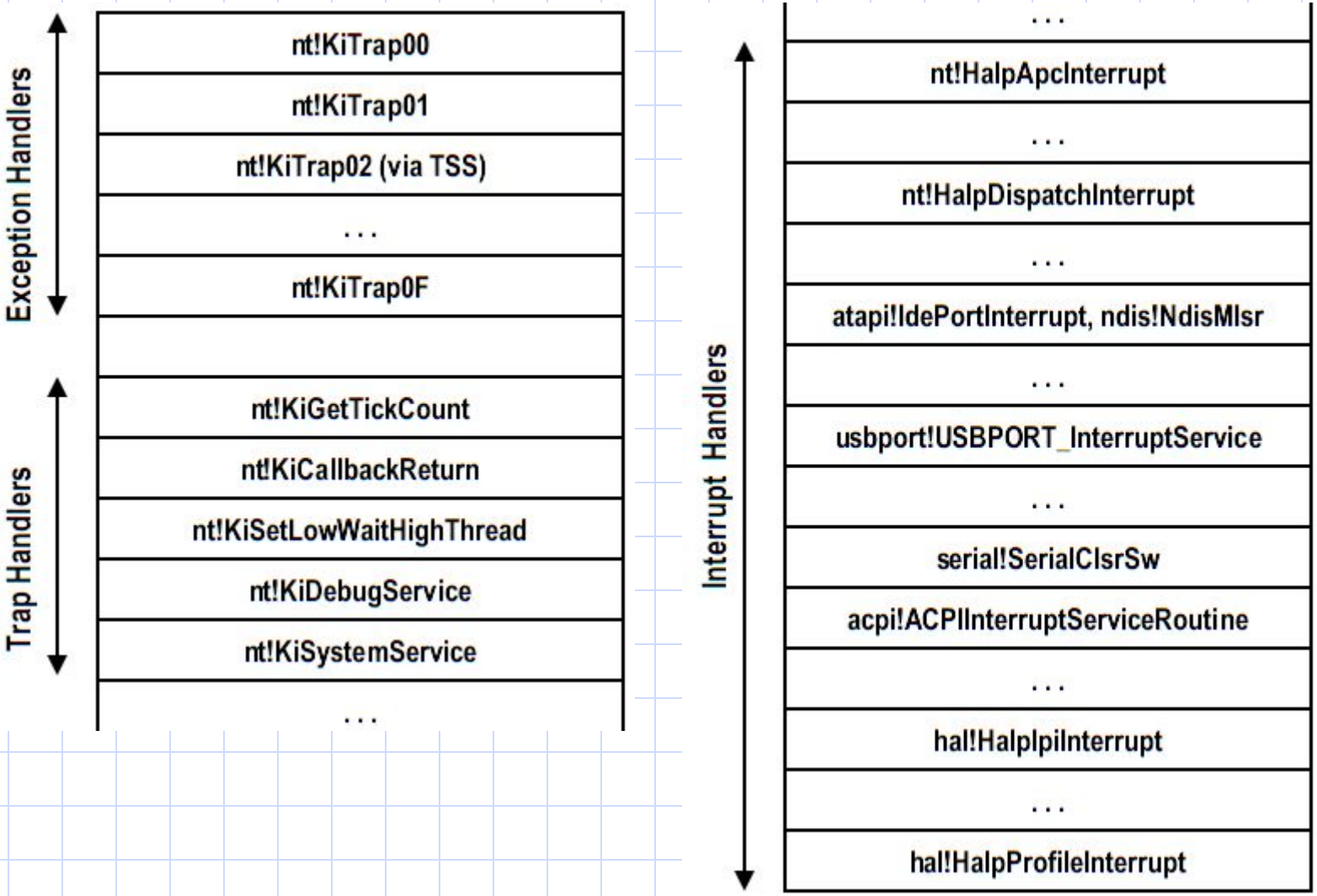


Механизм прерываний

- Почему нужны прерывания?
  - Существуют события, которые приостанавливают нормальное выполнение процессором кода приложения и заставляют ОС выполнить какие-то внеплановые действия. Они называются прерывания.
- По способу возникновения события бывают:
  - Внешние и внутренние;
  - Синхронные и асинхронные.
- Типы событий в классификации Windows:
  - Прерывание – внешнее или внутреннее асинхронное событие;
  - Исключение – внешнее или внутреннее синхронное событие;
  - Системный вызов – внутреннее синхронное событие.
- На все типы событий существует единая таблица векторов прерываний – Interrupt Dispatch Table (IDT):
  - Содержит адреса так называемых «ловушек» (Trap Handlers) для прерываний, исключений и системного вызова.
  - Размер таблицы ограничен 256 элементами.
  - Адрес этой таблицы хранится во внутреннем регистре процессора, который инициализируется при загрузке ОС.

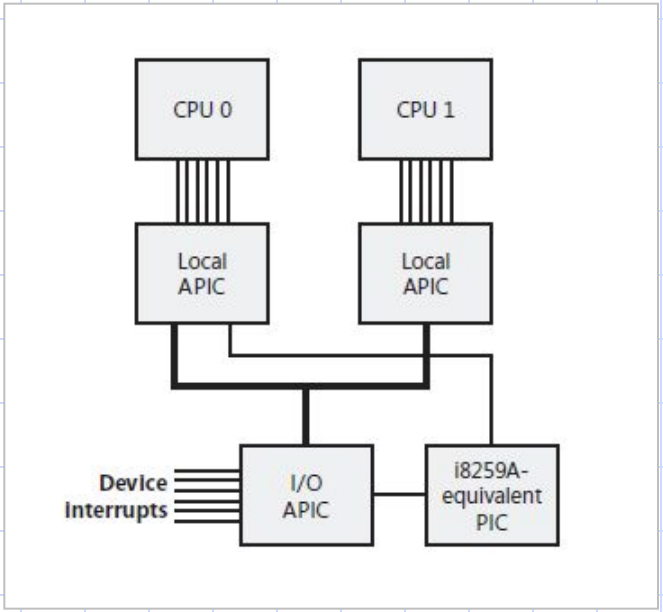
Таблица векторов прерываний (IDT)

- Таблица векторов прерываний – Interrupt Dispatch Table (IDT):



Аппаратные прерывания

- 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).



Приоритеты прерываний (IRQL)

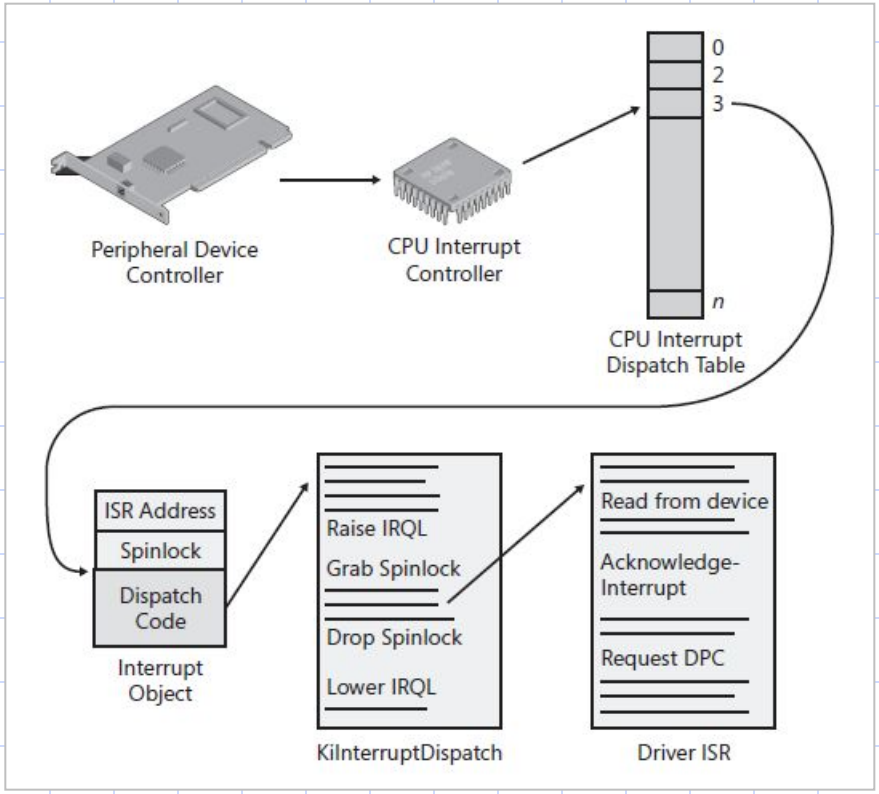
- Приоритеты прерываний – Interrupt Request Levels (IRQLs):
  - Прерывания обрабатываются в порядке приоритетов. Прерывание с более высоким приоритетом может прервать обработчик прерывания с более низким приоритетом. Все запросы на прерывание более низкого приоритета маскируются контроллером до завершения 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. Текущий Lazy IRQL процессора хранится в области управления процессором – Processor Control Region (PCR) и его расширенной части – Processor Control Block (PRCB). См. структуру KPCR.Prcb.
  - 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. В реализации функций KeRaiseIrql() и KeLowerIrql() используется оптимизация под названием Lazy IRQL.
  - Другие полезные функции IRQL API: **KeGetCurrentIrql**(), **KeRaiseIrqlToDpcLevel**().

Приоритеты прерываний для процессора x86

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

Процедура обработки прерываний (ISR)

- Процедура обработки прерываний – Interrupt Service Routine:
  - В таблице обработчиков прерываний хранятся указатели на так называемые объекты прерываний – Interrupt Objects. Каждый объект хранит данные и код (несколько инструкций процессора). Именно код зарегистрирован в IDT.



- Когда объект прерывания инициализируется, ОС создает его код из шаблона **KiInterruptTemplate**. Этот код вызывает одну из процедур ядра – KiInterruptDispatch или KiChainedDispatch, передавая ей объект прерывания.
- Процедура **KiInterruptDispatch** применяется, если для прерывания зарегистрирован один объект, **KiChainedDispatch** – несколько объектов. Если в цепочке объектов обработчик возвращает признак завершения обработки, следующие обработчики не вызываются.
- Чтобы подсоединить/отсоединить процедуру драйвера в качестве обработчика прерывания, вызывается функция **IoConnectInterrupt**/ **IoDisconnectInterrupt**.

Процедуры обработки прерываний

- Особенности процедур обработки прерываний:
  - При написании процедур обработки прерываний следует учитывать, что 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.
  - Отсюда следует, что в процедурах обработки аппаратных прерываний (и на уровне DISPATCH_LEVEL) можно работать лишь с физической памятью (non-paged memory). Объясняется это тем, что попытка доступа к странице, которой нет в памяти, вызывает прерывание, в ответ на которое менеджер памяти вынужден инициировать подкачку страницы с диска и 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.

Программные прерывания

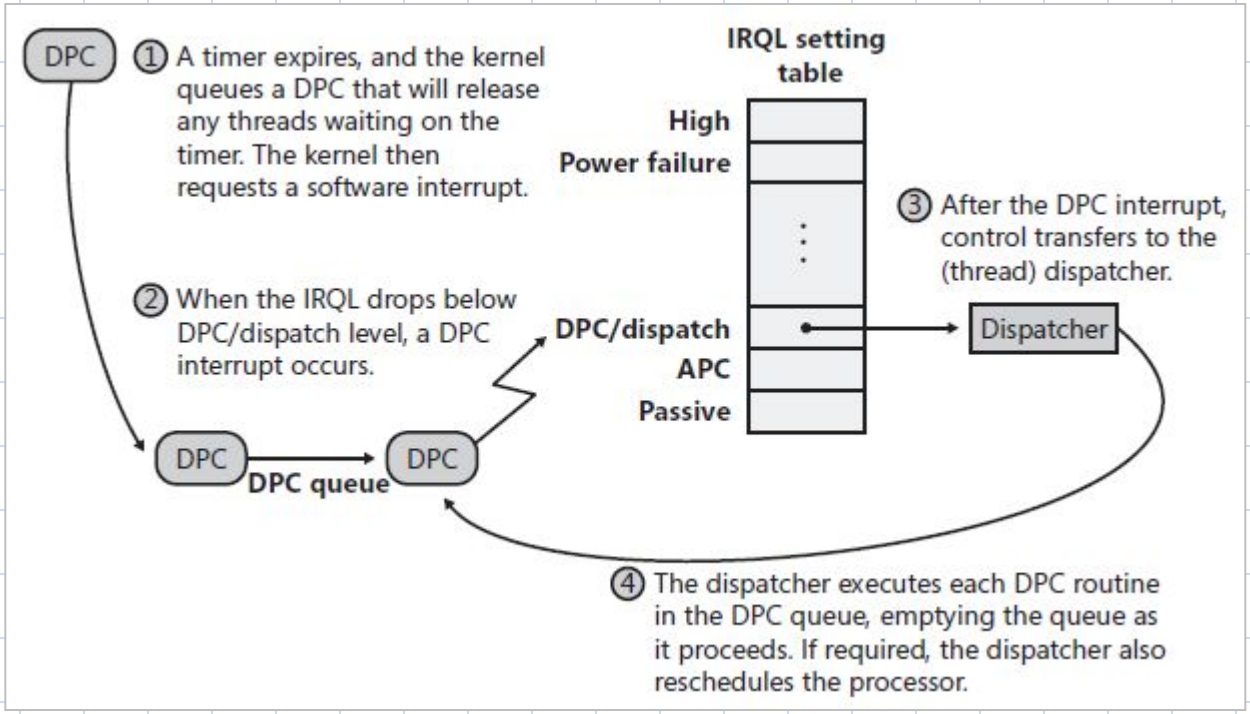
- Программные прерывания уровня 2:
  - Бывает ядро обнаруживает, что нужно выполнить переключение на другой поток, когда оно находится в глубине своего кода на уровне аппаратных прерываний (или на высшем уровне программных прерываний). В этом случае ядро запрашивает отложенное перепланирование. Когда обработка всех прерываний закончена, ядро проверяет, существуют ли отложенные запросы на переключение потока. Если есть, IRQL устанавливается в DISPATCH_LEVEL и выполняется обработка программного прерывания диспетчеризации потоков.
  - Аналогично может быть выполнена не только операция перепланирования, но и вызов другой системной функции. Этот вызов называется отложенным – Deferred Procedure Call (DPC).
  - И процедура перепланирования потоков, и отложенные процедуры совмещают один уровень приоритета прерываний номер 2. Поэтому этот уровень приоритета называется DISPATCH_LEVEL/DPC_LEVEL.

Отложенная процедура (DPC)

- Проблема задержки из-за обработки прерываний:
  - 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.
- Уменьшение задержки:
  - Процедура обработки прерываний выполняет абсолютный минимум работ с высоким уровнем приоритета;
  - Ставит в очередь к процессору отложенную процедуру для выполнения работы на уровне приоритета DPC_LEVEL/DIPATCH_LEVEL;
  - Позволяет системе быстро обработать другие прерывания.
- Отложенная процедура – Deferred Procedure Call (DPC):
  - Выполняется на уровне прерываний DPC_LEVEL/DIPATCH_LEVEL на том же процессоре, что и вызывающая ISR, или на заданном процессоре.
  - Использует для своего выполнения контекст произвольного потока.
  - На время работы подключает к процессору отдельный стек DPC-процедур.
  - При постановке DPC в очередь, указывается один из трех приоритетов: Low, Medium – в конец очереди, High – в начало очереди.

Отложенная процедура (DPC)

- Механизм обслуживания DPC-процедур:

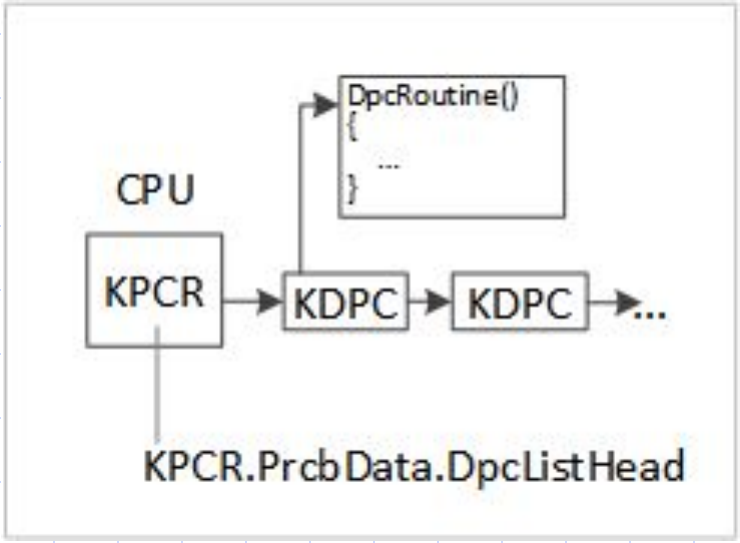


- Правила вызова прерываний для DPC-процедур:

DPC Priority	DPC Targeted at ISR's Processor	DPC Targeted at Another Processor
Low	DPC queue length exceeds maximum DPC queue length or DPC request rate is less than minimum DPC request rate	DPC queue length exceeds maximum DPC queue length or System is idle
Medium	Always	DPC queue length exceeds maximum DPC queue length or System is idle
High	Always	Always

Отложенная процедура (DPC)

- DPC-процедура в программе:
  - DPC-процедура представляется структурой ядра KDPC, в которой хранится адрес callback-процедуры, составляющей подпрограмму DPC.
  - Структура KDPC создается и ставится в очередь в специальный список отложенных процедур процессора. Он находится здесь: KPCR.PrcbData.DpcListHead.
  - Стек для выполнения DPC-процедур находится здесь: KPCR.PrcbData.DpcStack.
- Функции управления:
  - **KeInitializeDpc**()
  - **KeInsertQueueDpc**()
  - **KeRemoveQueueDpc**()
  - **KeSetTargetProcessorDpc**()
  - **KeSetImportanceDpc**()
  - **KeFlushQueuedDpcs**()



Асинхронная процедура (APC)

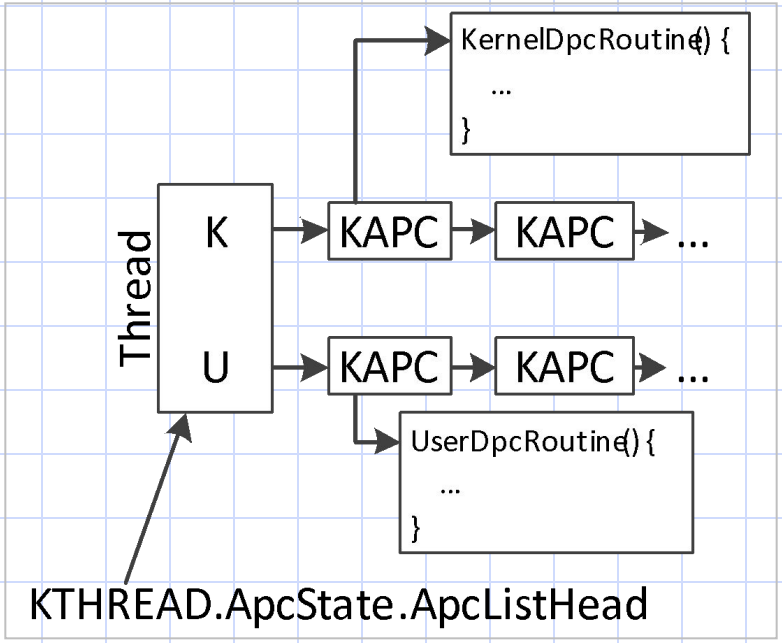
- Асинхронная процедура – Asynchronous Procedure Call (APC):
  - Как и DPC, применяется для выполнения отложенных действий.
  - Выполняется на уровне прерываний APC_LEVEL или PASSIVE_LEVEL в контексте заданного потока, и соответственно, в виртуальном адресном пространстве процесса, которому принадлежит поток.
  - APC-процедура не подвержена ограничениям DPC-процедур, она может захватывать объекты ядра, выполнять ожидания объектов, обращаться к отсутствующим страницам памяти, делать системные вызовы.
- Типы APC-процедур:
  - APC режима ядра – Kernel Mode APC, подразделяется на:
    - Специальную APC – Special Kernel Mode APC
    - Нормальную APC – Normal Kernel Mode APC
  - APC пользовательского режима – User Mode APC.

Асинхронная процедура (APC)

- APC-процедура в программе:
  - APC-процедура представляется структурой ядра KAPC. Структура KAPC содержит указатели на три подпрограммы:
    - RundownRoutine – выполняется, если из-за удаления потока удаляется структура KAPC.
    - KernelRoutine – выполняется на уровне приоритета APC_LEVEL.
    - NormalRoutine – выполняется на уровне приоритета PASSIVE_LEVEL.
  - Структура 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.

Асинхронная процедура (APC)

- Функции управления APC режима ядра:
  - **KeInitializeApc**()
  - **KeInsertQueueApc**()
  - **KeRemoveQueueApc**()
  - **KeFlushQueueApc**()
  - **KeAreApcsDisabled**()
  - **KeEnterGuardedRegion**()
  - **KeLeaveGuardedRegion**()
  - **KeEnterCriticalRegion**()
  - **KeLeaveCriticalRegion**()
- В пользовательском режиме:
  - **QueueUserApc**()



APC режима ядра

- 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.
  - Специальная APC 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.
  - Нормальная APC используется ОС для завершения обработки запроса от драйвера – Interrupt Request Packet (IRP).

APC пользовательского режима

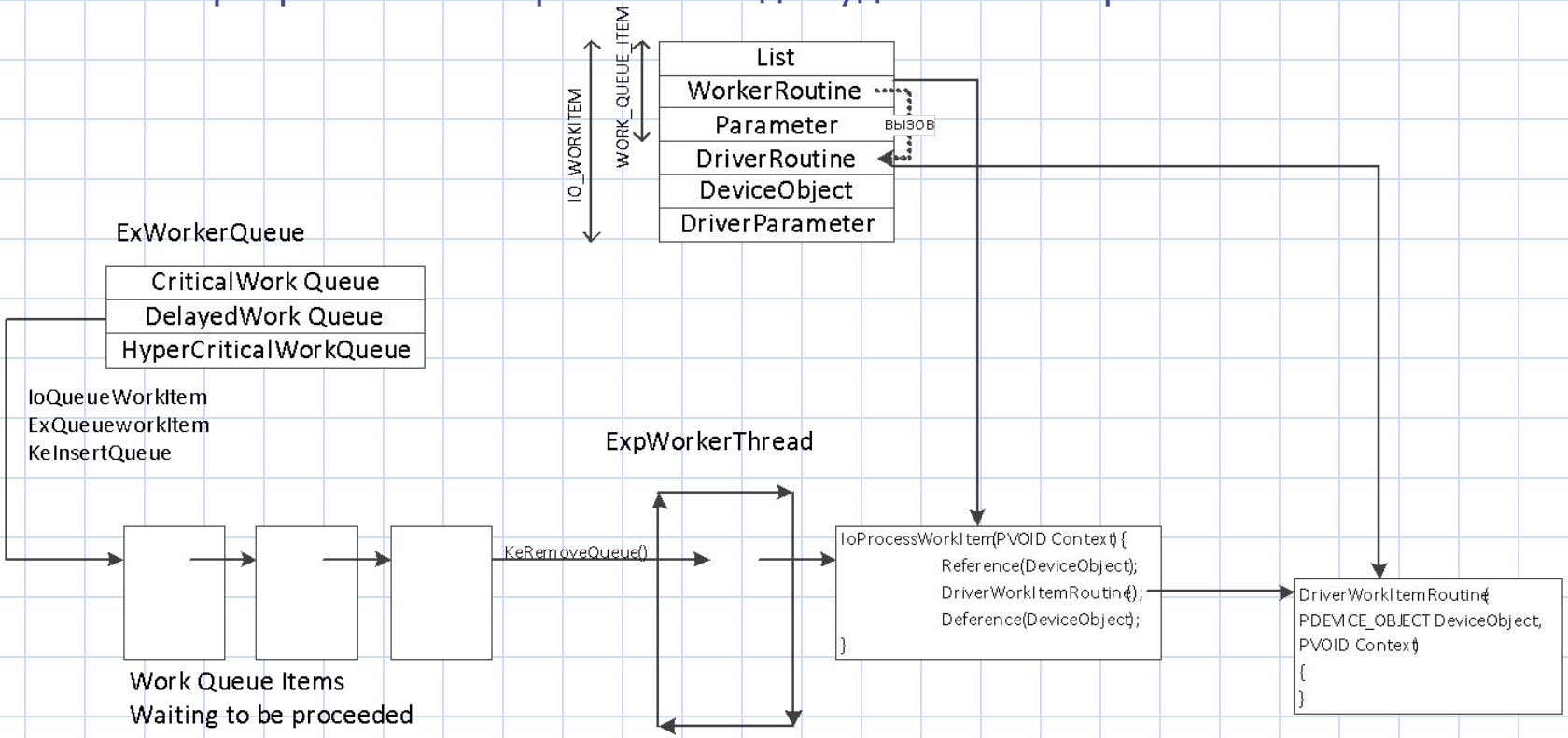
- APC-процедура пользовательского режима:
  - KAPC.NormalRoutine выполняется на уровне приоритета PASSIVE_LEVEL.
  - Вызывается, только если поток ожидает объект ядра в alertable-режиме, например WaitForSingleObjectEx(…, true), WaitForMultipleObjectsEx(…, true), SleepEx(…, true).
  - Создается вызовом 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.

Элемент работы (Work Item)

- Элемент работы – Work Item:
  - Механизм выполнения асинхронных действий в контексте системного потока на уровне приоритета PASSIVE_LEVEL.
  - 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.

Очередь элементов работы

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



Управление памятью

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

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).

Оптимизация работы кучи

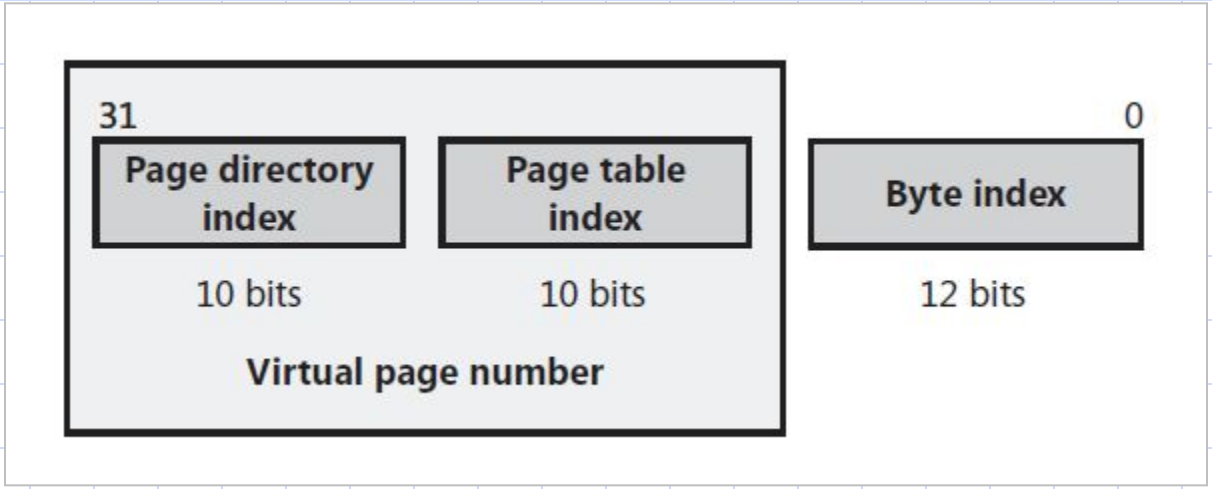
- Списки предыстории – Look-aside Lists:
  - Применяются менеджером кучи для выделения-освобождения элементов фиксированного размера. В ядре могут явно применяться драйверами.
  - Представлены в виде 128 связных списков свободных блоков. Каждый список содержит элементы строго определенного размера – от 8 байтов до 1 КБ на x86 и от 16 байтов до 2 КБ на x64.
  - Когда блок памяти освобождается, он помещается в список предыстории, соответствующий его размеру. Затем, если запрашивается блок памяти такого же размера, он берется из списка предыстории методом LIFO. Для организации списка предыстории используются функции **InterlockedPushEntrySList**() и **InterlockedPopEntrySList**().
  - Раз в секунду ОС уменьшает глубину списков предыстории с помощью функции ядра **KiAdjustLookasideDepth**().
- Низко-фрагментированная куча – Low Fragmentation Heap:
  - Включается с помощью **HeapSetInformation**().
  - Уменьшает фрагментацию памяти за счет хранения в списках предыстории элементов одного размера вместе.
  - Улучшает масштабируемость на многопроцессорных системах путем поддержания количества внутренних структур данных равным количеству процессоров в системе, умноженному на 2.

Виртуальная память

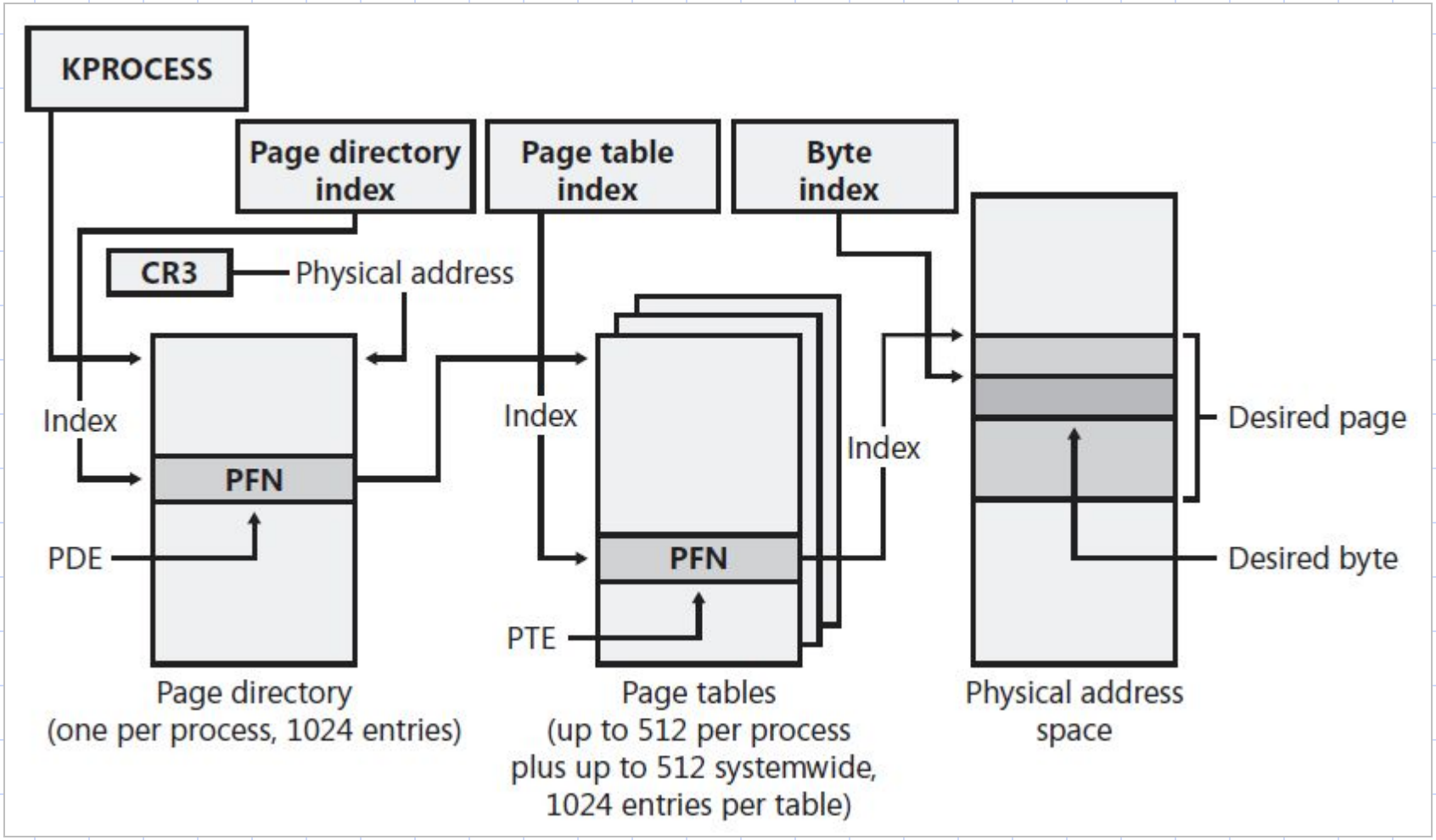
- Виртуальная память состоит из двух видов страниц:
  - Малые страницы – 4 КБ.
  - Большие страницы – 4 МБ на x86 или 2 МБ на x64. Большие страницы используются для Ntoskrnl.exe, Hal.dll, данных ядра, описывающих резидентную память ядра и состояние физических страниц памяти. При вызове VirtualAlloc() можно указать флаг MEM_LARGE_PAGE. Для всей страницы применяется единый режим защиты и блокировки памяти.
- Страница памяти может находиться в одном из 3-х состояний:
  - Отсутствует – Free. Обращение к адресу в такой странице приводит к сбою.
  - Передана (в физическую память) – Committed. Обращение к адресу в такой странице транслируется в физический адрес памяти.
  - Зарезервирована (в таблице страниц) – Reserved. Обращение к адресу в такой странице может быть обработано системой, которая передаст процессу реальную страницу виртуальной памяти. Используется для резервирования непрерывного диапазона виртуальных адресов. Пример – стек потока. Он состоит из одной переданной в физическую память страницы и набора зарезервированных страниц стандартным общим размером 1 МБ.

Виртуальная память

- Структура виртуального адреса:
  - Номер таблицы страниц в каталоге таблиц – Page directory index -> Page Directory Entry.
  - Номер страницы в таблице страниц – Page table index -> Page Table Entry.
  - Смещение в странице – Byte index.



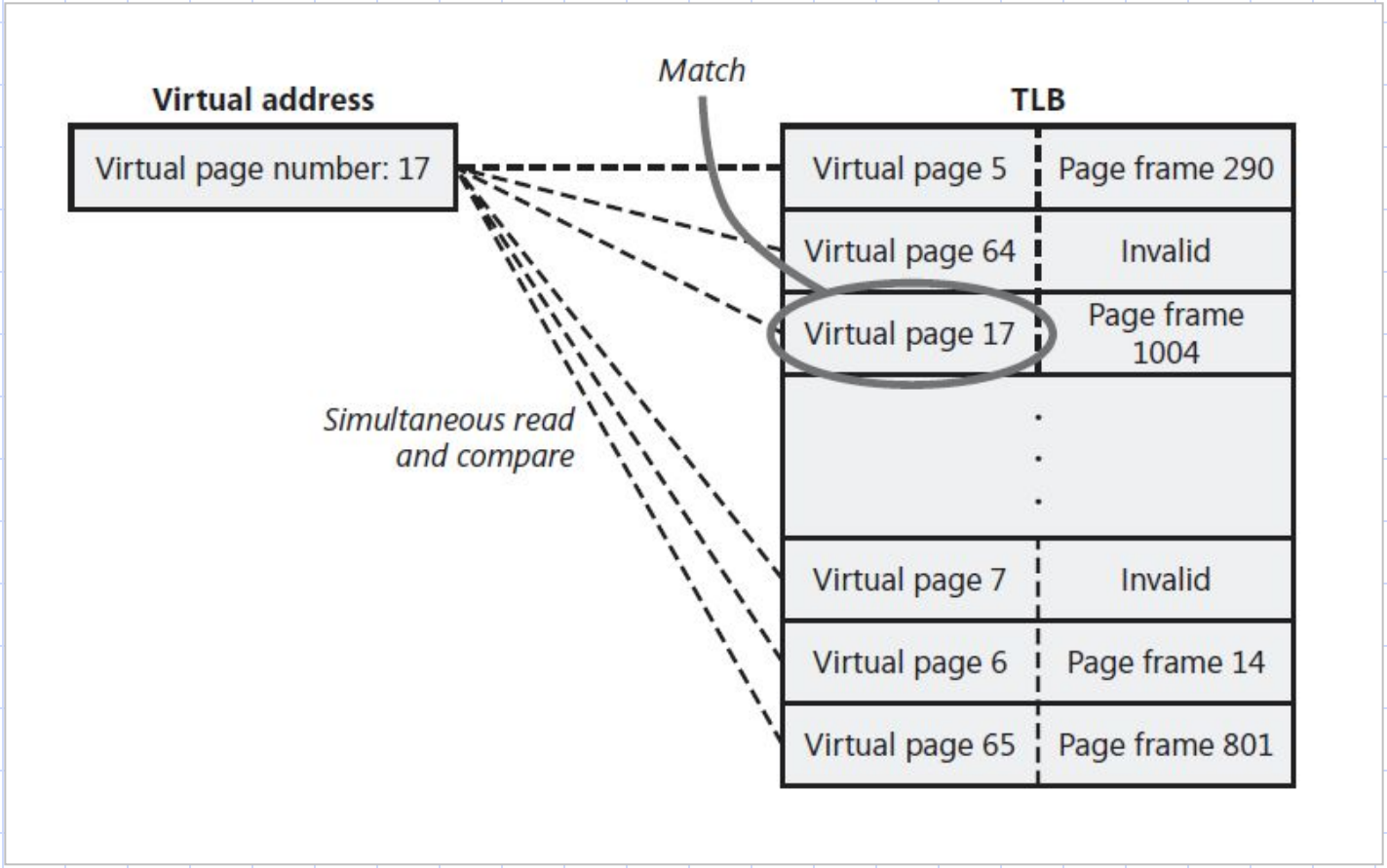
Трансляция виртуального адреса в физический



- CR3 – регистр процессора, хранящий физический адрес каталога таблиц.
- PFN – Page Frame Number – номер страничного кадра.

Кэширование виртуальных адресов

- Кэширование виртуальных адресов существенно повышает производительность процессора при работе с памятью.
- Кэш называется Translation Look-aside Buffer (TLB).



Управление памятью в режиме ядра

- Пулы памяти – Memory Pools
- Списки предыстории – Look-aside Lists
- Представление объектов ядра в памяти
- Фиксация данных в физической памяти
- Таблицы описания памяти – Memory Descriptor Lists

Пулы памяти

- Пулы памяти – Memory Pools:
  - 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.
  - Дополнительно операционная система создает и поддерживает:
    - Страничный пул сеансовой памяти (Session Pool).
    - Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти.
    - 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.
  - Резидентный и страничные пулы растут до установленного максимума:
    - Non-Paged Pool – 256 МБ на x86, 128 ГБ на x64.
    - Paged Pool – 2 ГБ на x86, 128 ГБ на x64.

Пулы памяти

- 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".
- Освобождение памяти в пуле:
  - void **ExFreePoolWithTag**(void* P, ULONG Tag);
  - **ExFreePool**().
- Конфигурация предельных размеров пулов:
  - HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management
  - NonPagedPoolQuota – квота резидентного пула для процесса (<128 МБ).
  - NonPagedPoolSize – максимальный размер резидентного пула.
  - PagedPoolQuota – квота страничного пула для процесса (<128 МБ).
  - PagedPoolSize – максимальный размер страничного пула.
  - SessionPoolSize – максимальный размер страничного сеансового пула.

Описатель пула памяти

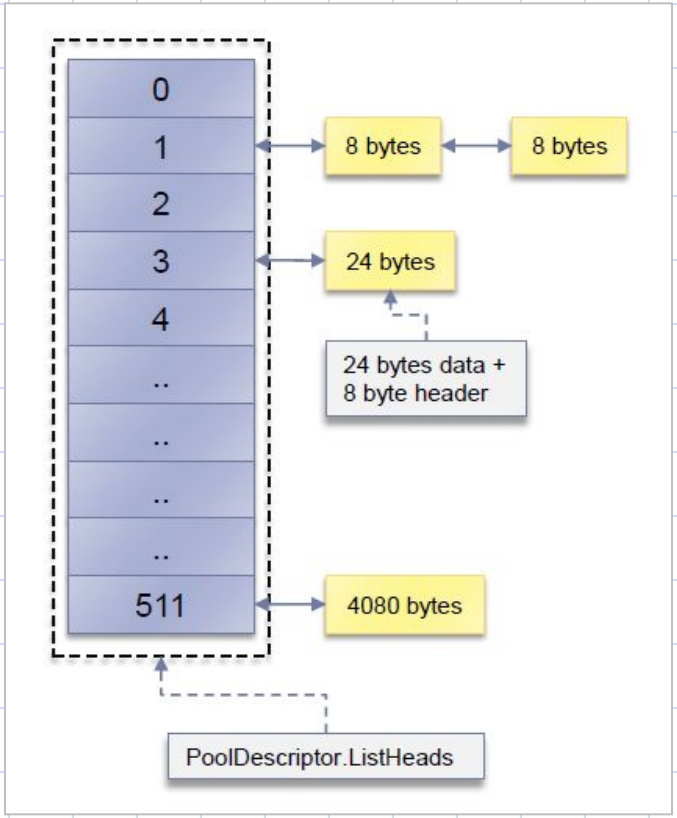
- Каждый пул описывается структурой POOL_DESCRIPTOR (Windows 7):
  - struct POOL_DESCRIPTOR
    {
    POOL_TYPE PoolType;
    KGUARDED_MUTEX PagedLock;
    ULONG NonPagedLock;
    LONG RunningAllocs;
    LONG RunningDeAllocs;
    LONG TotalBigPages;
    LONG ThreadsProcessingDereferrals;
    ULONG TotalBytes;
    ULONG PoolIndex;
    LONG TotalPages;
    VOID** **PendingFrees**;
    LONG PendingFreeDepth;
    LIST_ENTRY **ListHeads**[512];
    };

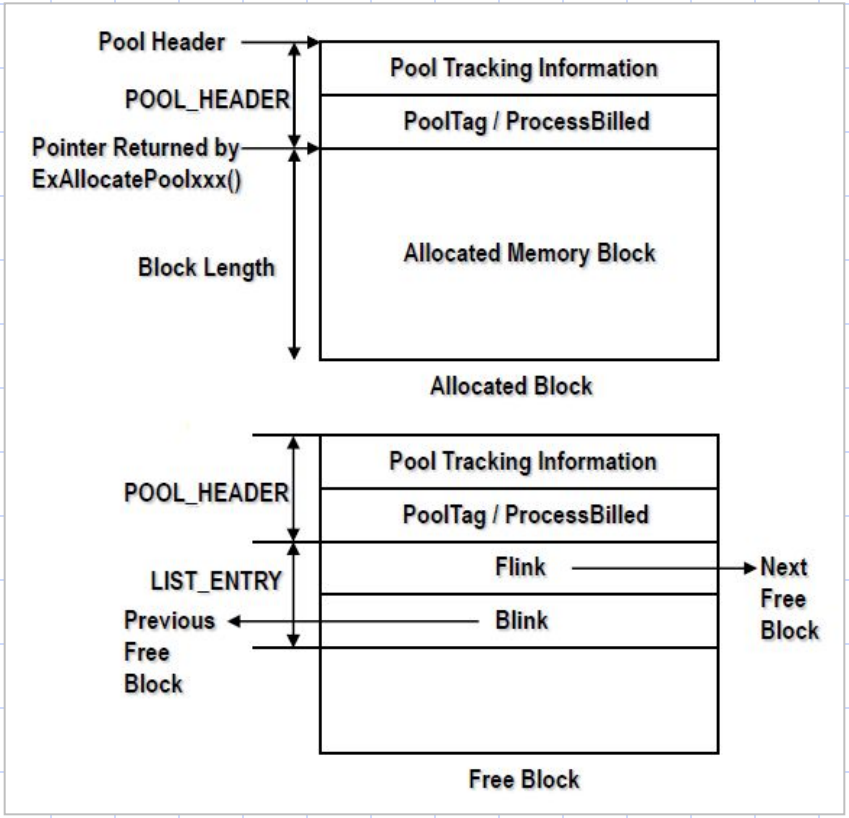
Доступ к описателям пулов памяти

- 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 пять описателей страничных пулов. Доступ к ним можно получить через переменную nt!ExpPagedPoolDescriptor. Это массив из 5 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.

Список свободных блоков пула (x86)

- 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.



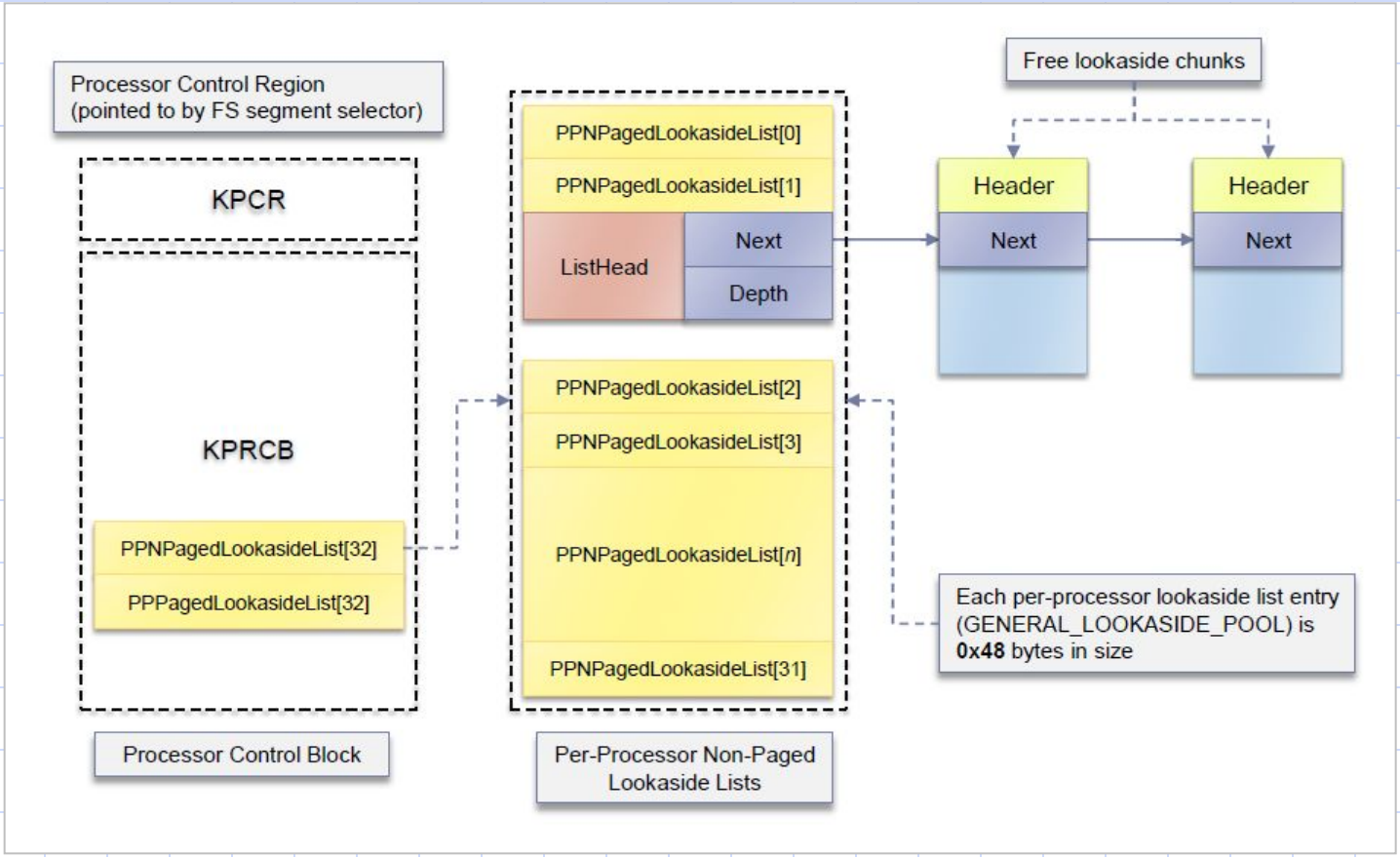


Заголовок блока пула

- 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).

Списки предыстории – Look-aside Lists

- На каждый процессор создается набор списков предыстории:
  - 32 списка предыстории на процессор.
  - В каждом списке – свободные блоки строго определенного размера от 8 до 256 байтов с гранулярностью 8 (x86).

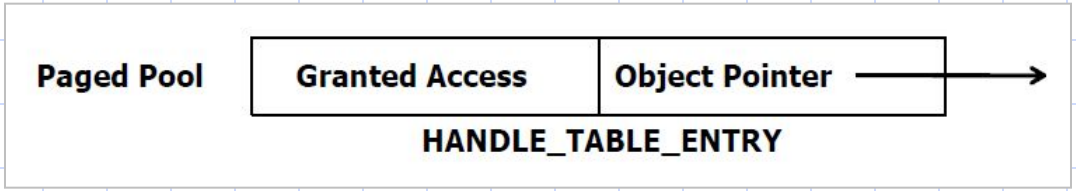


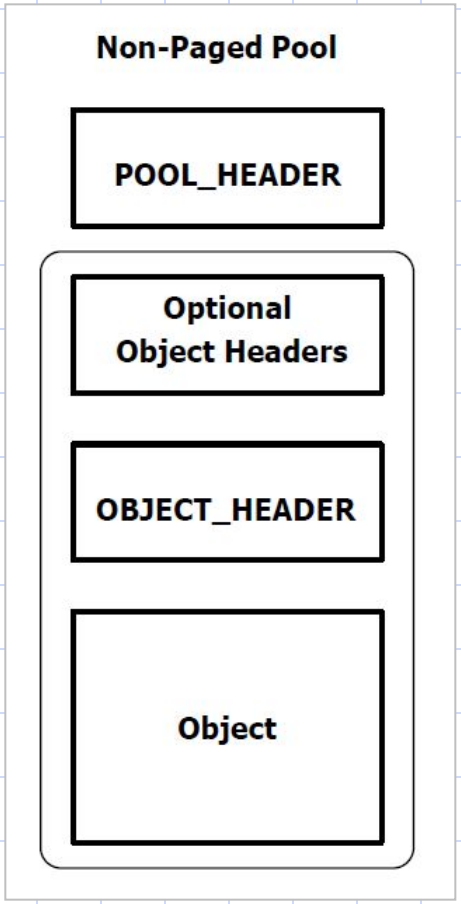
Списки предыстории – Look-aside Lists

- 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.

Представление объектов ядра в памяти

- 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.





Фиксация данных в физической памяти

- 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 (MDL)

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

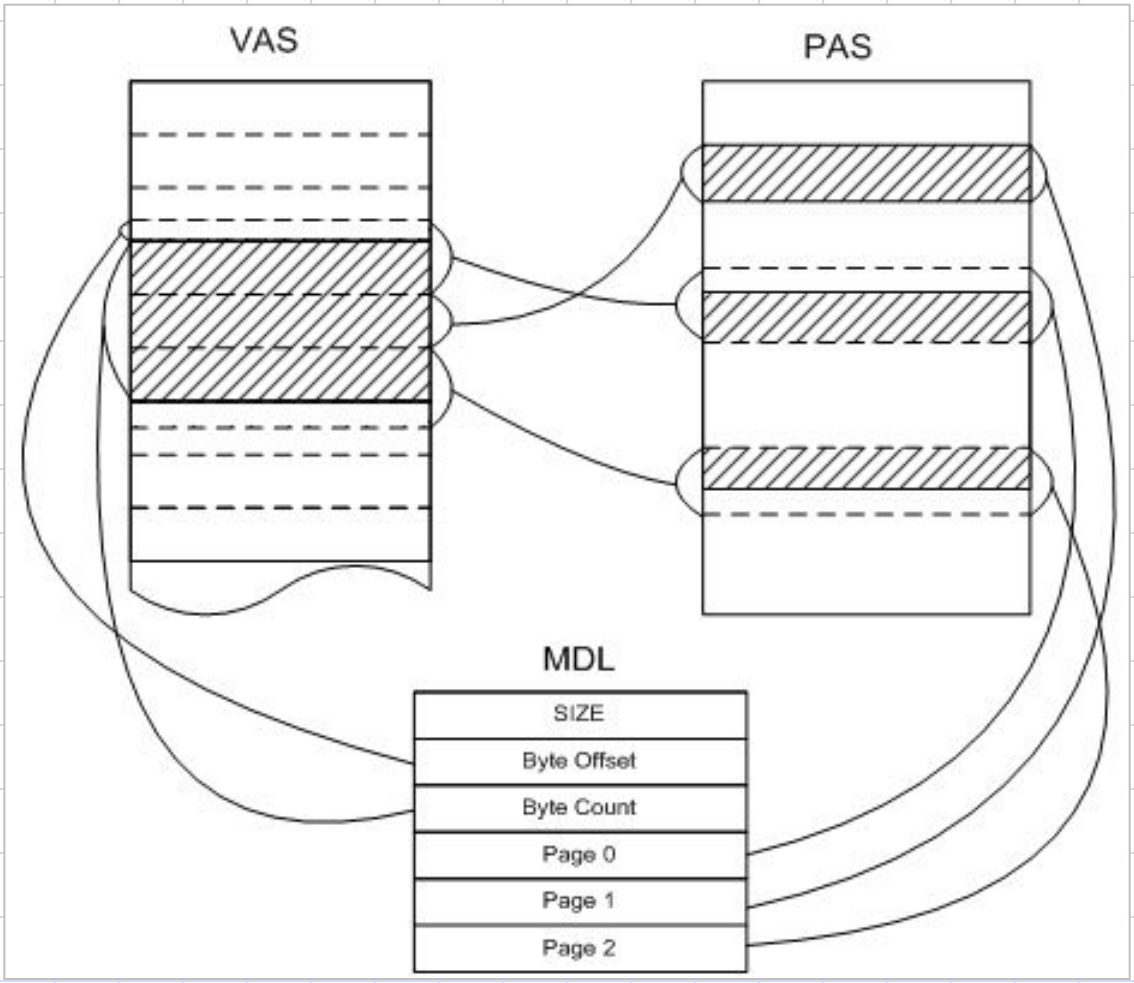
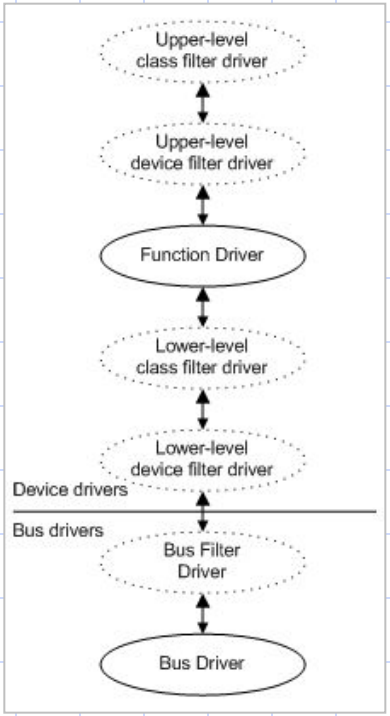


Таблица описания памяти (MDL)

- Заголовок MDL, за которым следует массив номеров страниц:
  - struct MDL
    {
    PMDL Next; // Цепочка буферов. Можно обратиться напрямую.
    SHORT Size; // Размер структуры, включает массив номеров страниц.
    SHORT MdlFlags; // Флаги. Можно обратиться напрямую.
    PEPROCESS Process; // Процесс, из квоты которого выделили память.
    PVOID MappedSystemVa; // Виртуальный адрес системной памяти.
    PVOID StartVa; // Виртуальный адрес буфера (page aligned).
    ULONG ByteCount; // Размер буфера.
    ULONG ByteOffset; // Смещение до буфера относительно StartVa.
    };
  - 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()

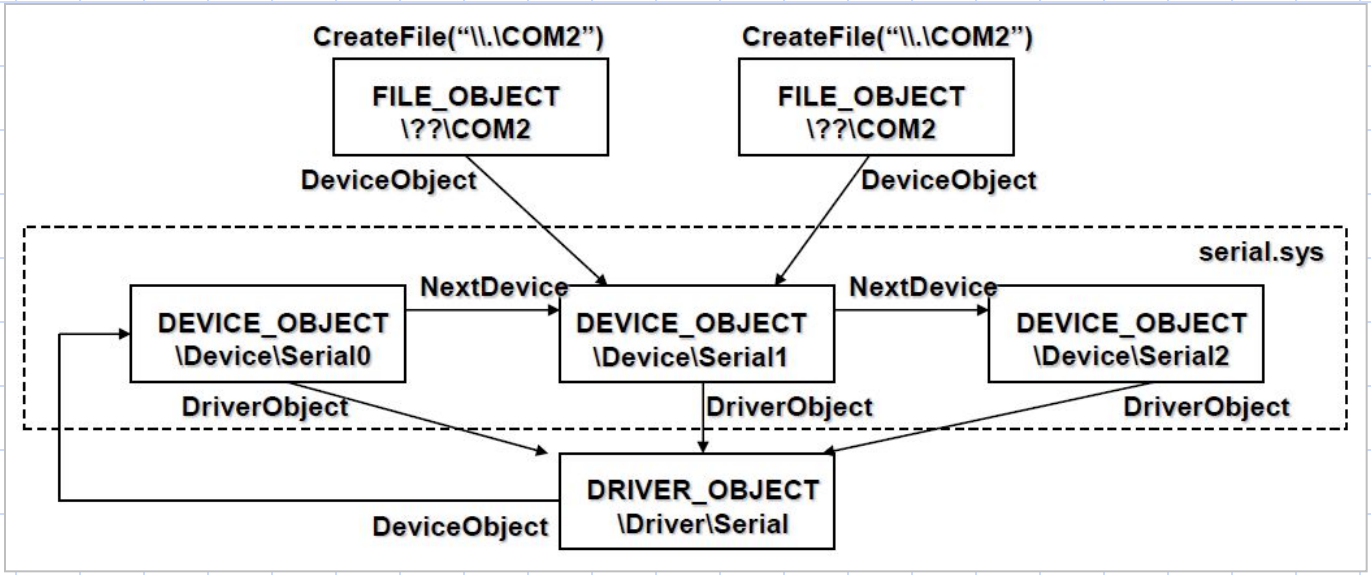
Драйверы Windows

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

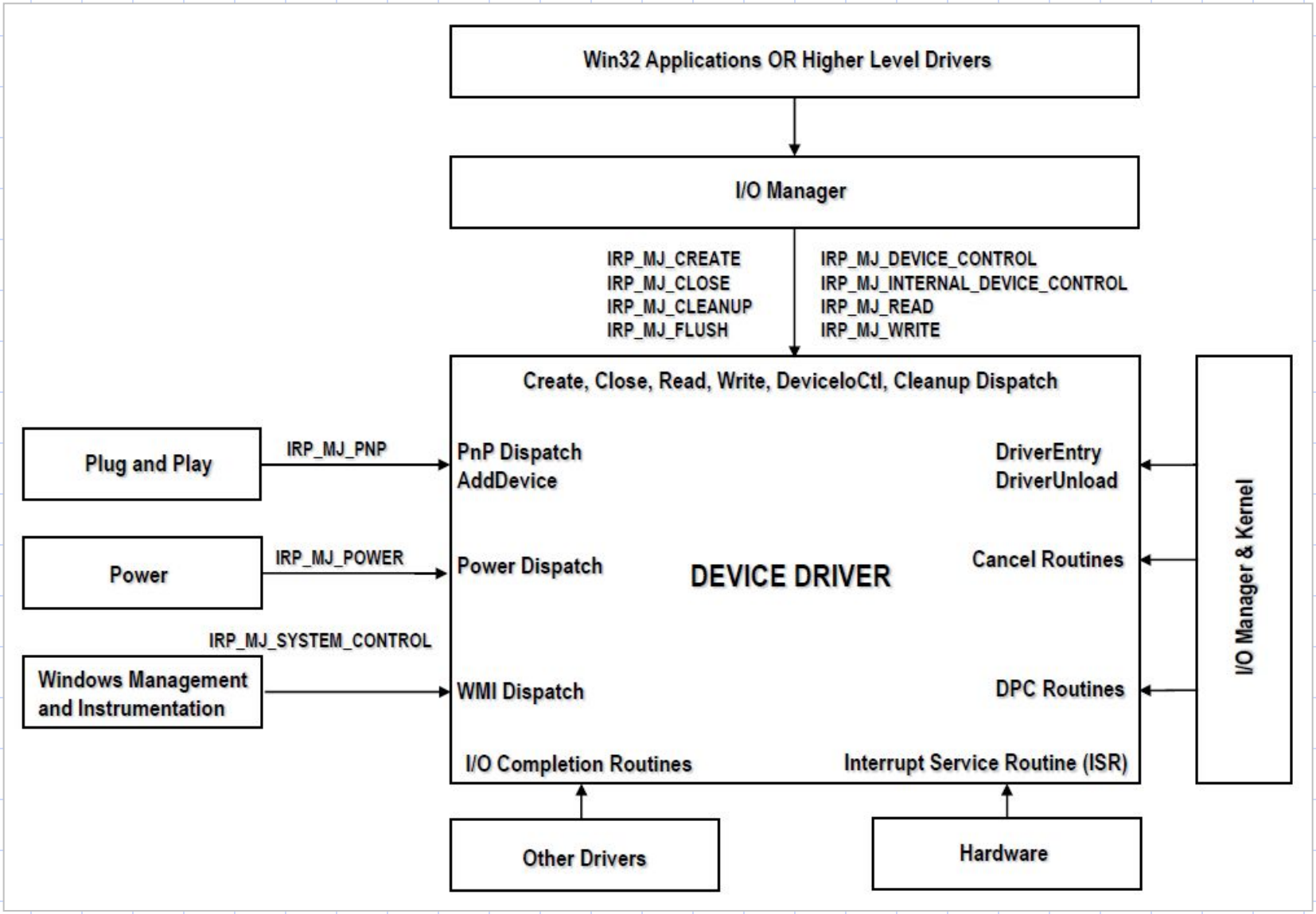


Объекты в драйвере

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



Точки входа в драйвер



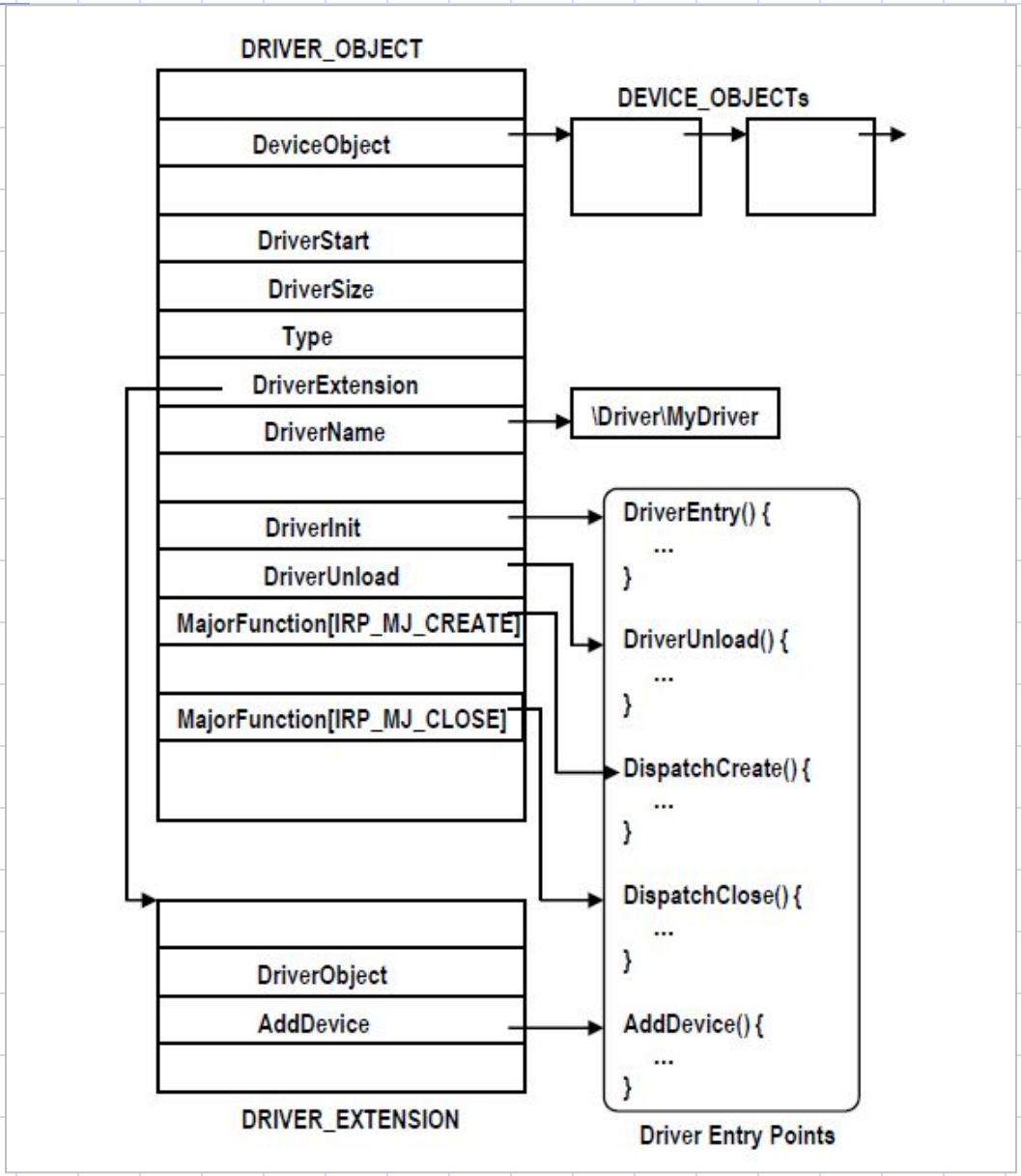
Главная точка входа в драйвер – DriverEntry

- 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 DRIVER_OBJECT:
  - Представляет загруженный в память драйвер.
  - Создается в единственном экземпляре в момент загрузки модуля драйвера в память операционной системой. Уничтожается в момент выгрузки модуля драйвера из памяти.
  - Передается в главную точку входа DriverEntry и процедуру XxxAddDevice.
  - Хранит состояние, общее для всех обслуживаемых драйвером устройств.
- Содержит:
  - Имя драйвера в структуре имен операционной системы.
  - Начальный адрес и размер драйвера в памяти.
  - Таблицу точек входа в драйвер.
  - Указатель на область расширенных данных драйвера, в которой хранится точка входа в процедуру XxxAddDevice.
  - Список созданных драйвером объектов DEVICE_OBJECT, представляющих обслуживаемые драйвером устройства.

Объект DRIVER_OBJECT

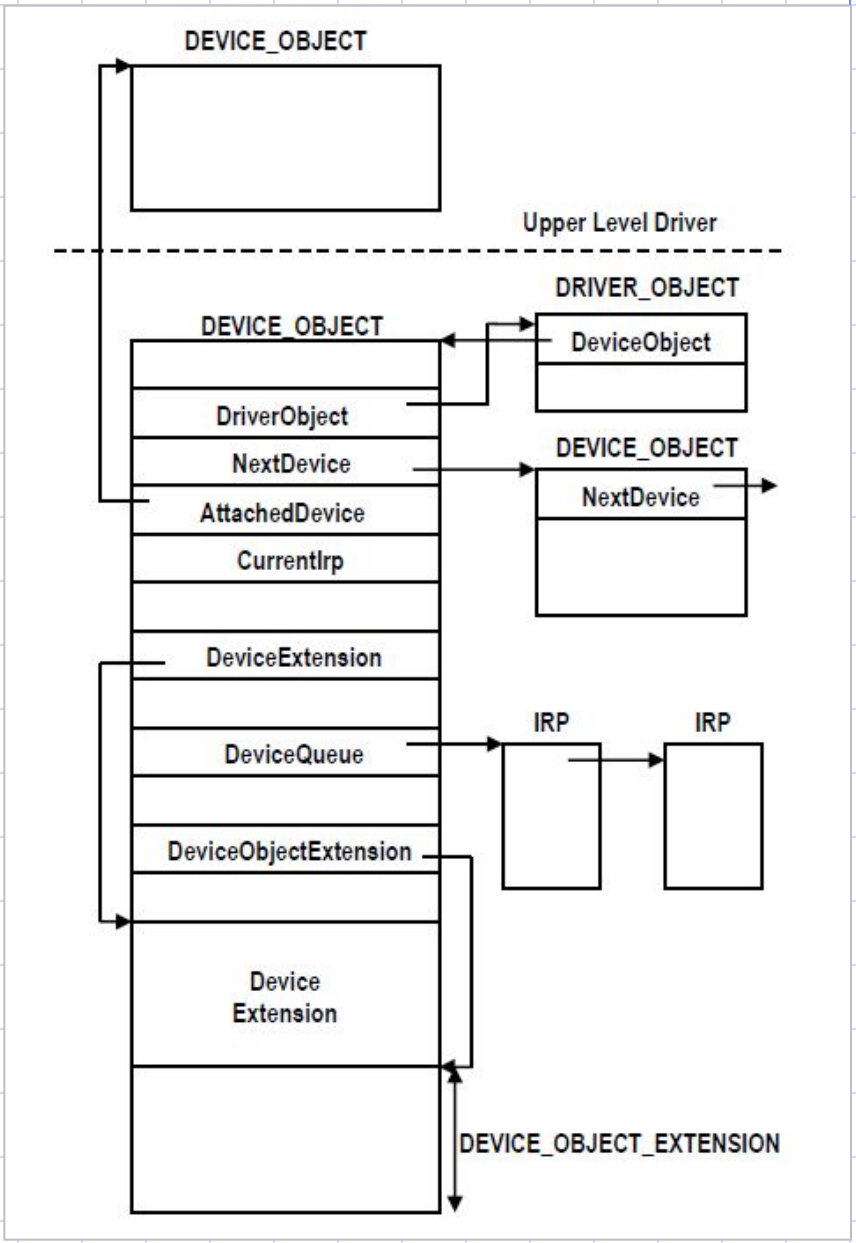


Объект DEVICE_OBJECT

- Объект DEVICE_OBJECT:
  - Представляет физическое или логическое устройство.
  - Создается драйвером с помощью процедуры **IoCreateDevice**(). Уничтожается с помощью процедуры **IoDeleteDevice**().
  - Передается в процедуру XxxAddDevice.
  - Может не иметь имени. Тогда оно автоматически генерируется операционной системой – FILE_AUTOGENERATED_DEVICE_NAME.
- Содержит:
  - Фиксированную (DEVICE_OBJECT) и переменную часть данных устройства. Размер и содержимое переменной части определяются драйвером.
  - Указатель на область расширенных данных объекта – DEVOBJ_EXTENSION. Таким образом, расширений получается два.
  - Указатель на владельца – объект драйвера – DriverObject.
  - Указатель на такой же объект устройства в драйвере верхнего уровня – AttachedDevice. Получающийся список образует стек драйверов.
  - Указатель на следующее устройство в списке драйвера – NextDevice.
  - Указатель на очередь I/O-запросов к устройству – DeviceQueue, и текущий запрос к устройству – CurrentIrp.

Объект DEVICE_OBJECT

```
struct DEVICE_OBJECT
{
    CSHORT Type;
    USHORT Size;
    LONG ReferenceCount;
    DRIVER_OBJECT* DriverObject;
    DEVICE_OBJECT* NextDevice;
    DEVICE_OBJECT* AttachedDevice;
    IRP* CurrentIrp;
    IO_TIMER* Timer;
    ULONG Flags;
    ULONG Characteristics;
    VPB* Vpb; // volume parameter block
    VOID* DeviceExtension;
    DEVICE_TYPE DeviceType;
    CCHAR StackSize;
    union { LIST_ENTRY ListEntry;
        WAIT_CONTEXT_BLOCK Wcb; } Queue;
    ULONG AlignmentRequirement;
    KDEVICE_QUEUE DeviceQueue;
    KDPC Dpc;
    ULONG ActiveThreadCount;
    SECURITY_DESCRIPTOR* SecurityDescriptor;
    KEVENT DeviceLock;
    USHORT SectorSize;
    USHORT Spare1;
    DEVOBJ_EXTENSION* DeviceObjectExtension;
    VOID* Reserved;
};
```

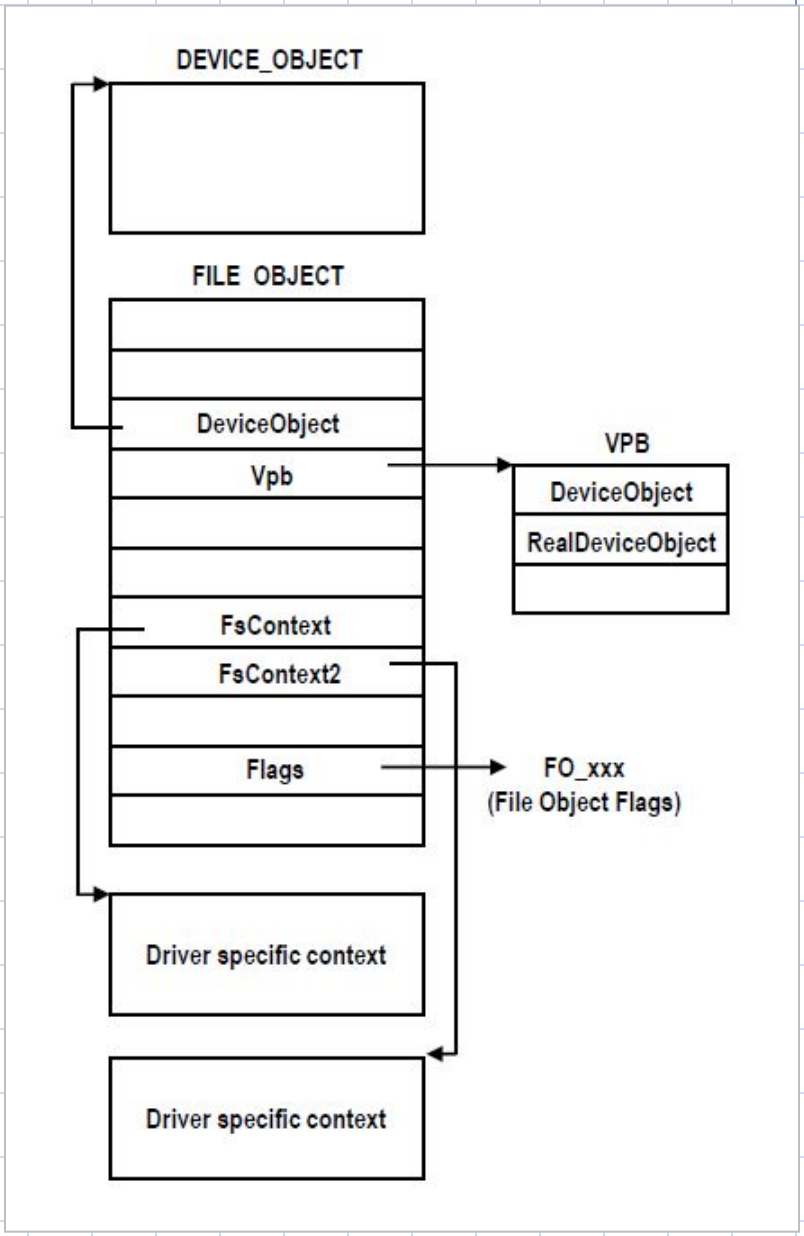


Объект FILE_OBJECT

- Объект FILE_OBJECT:
  - Представляет файл, открытый на устройстве.
  - Создается вызовом **CreateFile**()/**ZwCreateFile**().
  - Удаляется вызовом **CloseHandle**()/**ZwClose**().
- Содержит:
  - Указатель на владельца – объект устройства – DeviceObject.
  - Относительное имя файла, интерпретируемое драйвером устройства или драйвером файловой системы, – FileName.
  - Дополнительные данные, необходимые драйверам для работы с файлом, – FsContext и FsContext2.
  - Указатель на блок параметра тома, устанавливающий соответствие между файловой системой и смонтированным томом на устройстве, – Vpb.
  - Объект синхронизации Event, который блокирует потоки, осуществляющие синхронные запросы к устройству. Обработка запросов выполняется асинхронно.

Объект FILE_OBJECT

```
struct FILE_OBJECT
{
    CSHORT Type;
    CSHORT Size;
    DEVICE_OBJECT* DeviceObject;
    VPB* Vpb; // volume parameter block
    VOID* FsContext;
    VOID* FsContext2;
    SECTION_OBJECT_POINTERS* SectionObjectPointer;
    VOID* PrivateCacheMap;
    NTSTATUS FinalStatus;
    FILE_OBJECT* RelatedFileObject;
    BOOLEAN LockOperation; BOOLEAN DeletePending;
    BOOLEAN ReadAccess; BOOLEAN WriteAccess;
    BOOLEAN DeleteAccess; BOOLEAN SharedRead;
    BOOLEAN SharedWrite; BOOLEAN SharedDelete;
    ULONG Flags;
    UNICODE_STRING FileName;
    LARGE_INTEGER CurrentByteOffset;
    ULONG Waiters;
    ULONG Busy;
    VOID* LastLock;
    KEVENT Lock;
    KEVENT Event;
    IO_COMPLETION_CONTEXT* CompletionContext;
    KSPIN_LOCK IrpListLock;
    LIST_ENTRY IrpList;
    VOID* FileObjectExtension;
};
```



Пакет ввода-вывода (IRP)

- Пакет ввода-вывода – Input-Output Request Packet (IRP):
  - 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**().
- Содержит:
  - 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.

Пакет ввода-вывода (IRP)

```
struct IRP
{
    PMDL  MdlAddress;
    ULONG  Flags;
    union { . . .
        IRP* MasterIrp;
        VOID* SystemBuffer;
    } AssociatedIrp;
    IO_STATUS_BLOCK  IoStatus;
    KPROCESSOR_MODE  RequestorMode;
    BOOLEAN PendingReturned;
    BOOLEAN Cancel;
    KIRQL CancelIrql;
    DRIVER_CANCEL* CancelRoutine;
    VOID* UserBuffer;
    union { . . .
        struct { . . .
            union { KDEVICE_QUEUE_ENTRY
                DeviceQueueEntry;
                struct {
                    PVOID  DriverContext[4];
                };
            };
            ETHREAD* Thread;
            LIST_ENTRY  ListEntry;
        } Overlay;
    } Tail;
    . . .
};
```

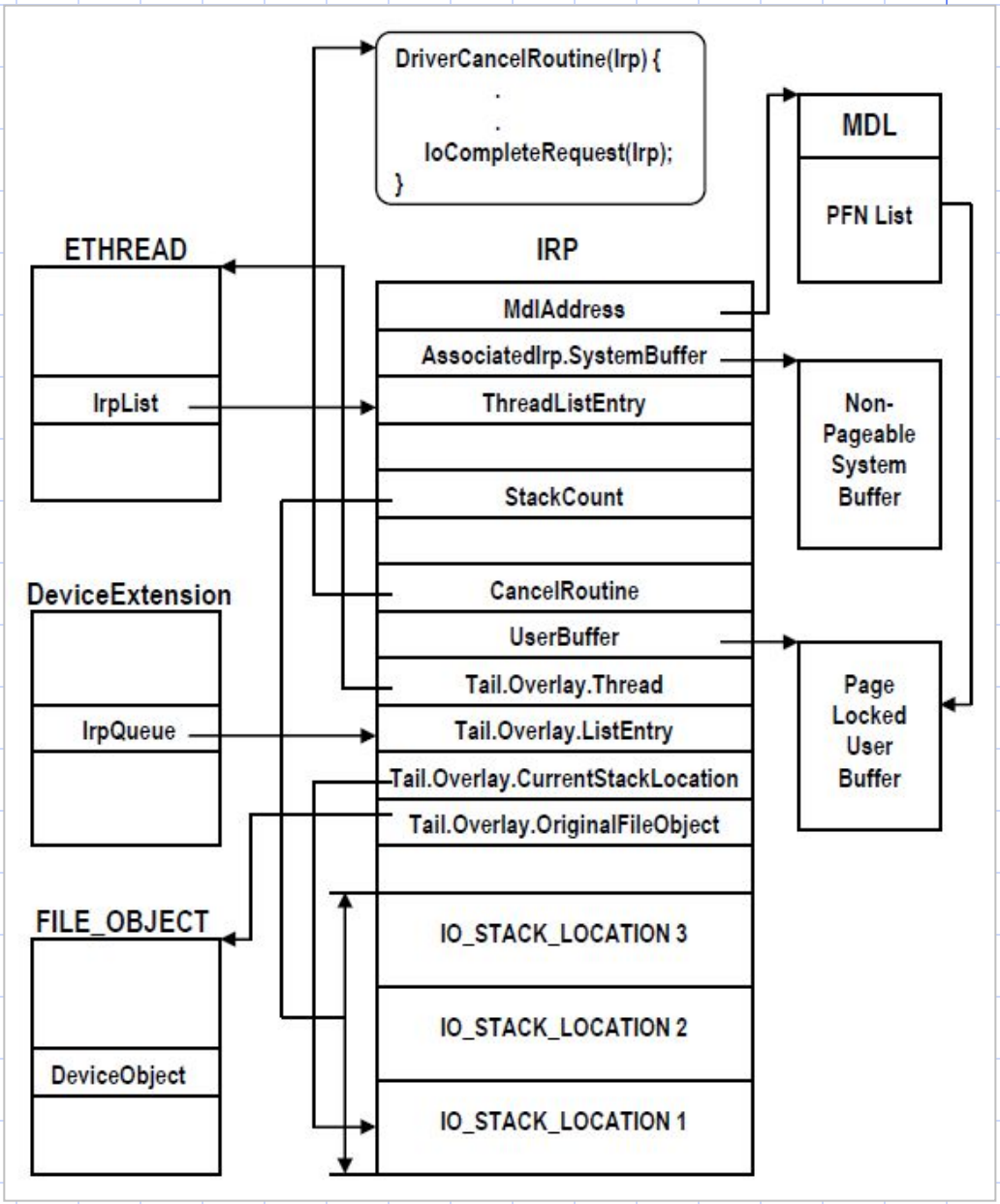
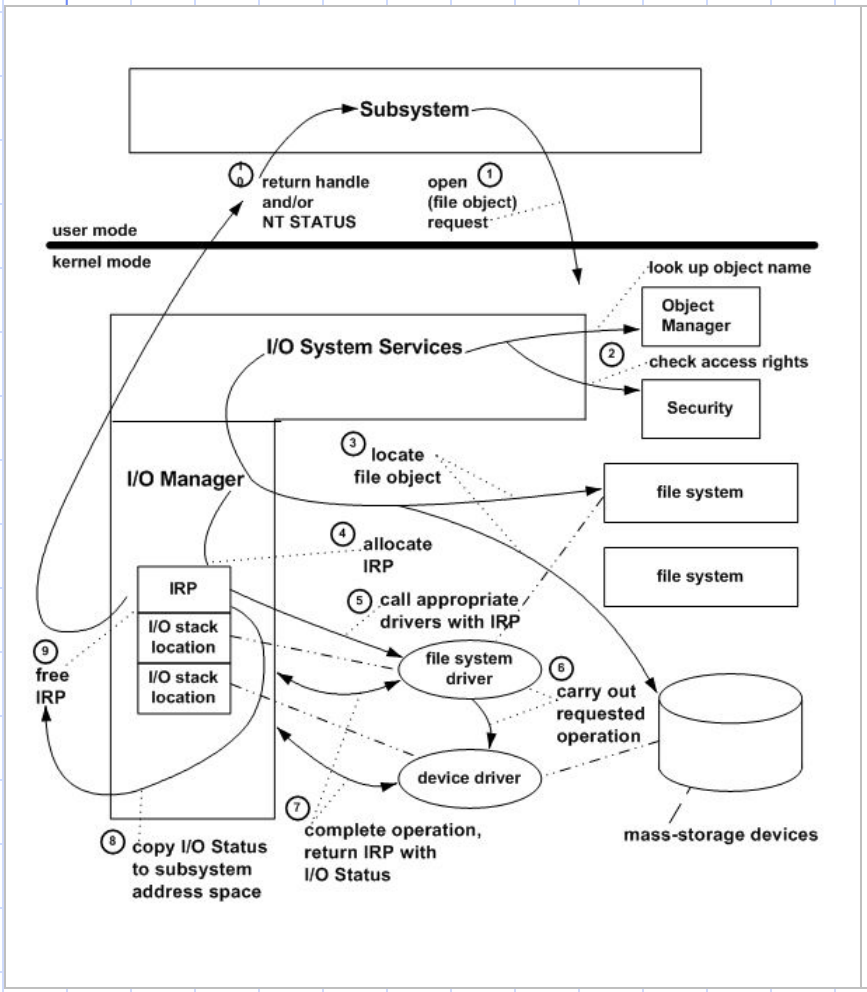
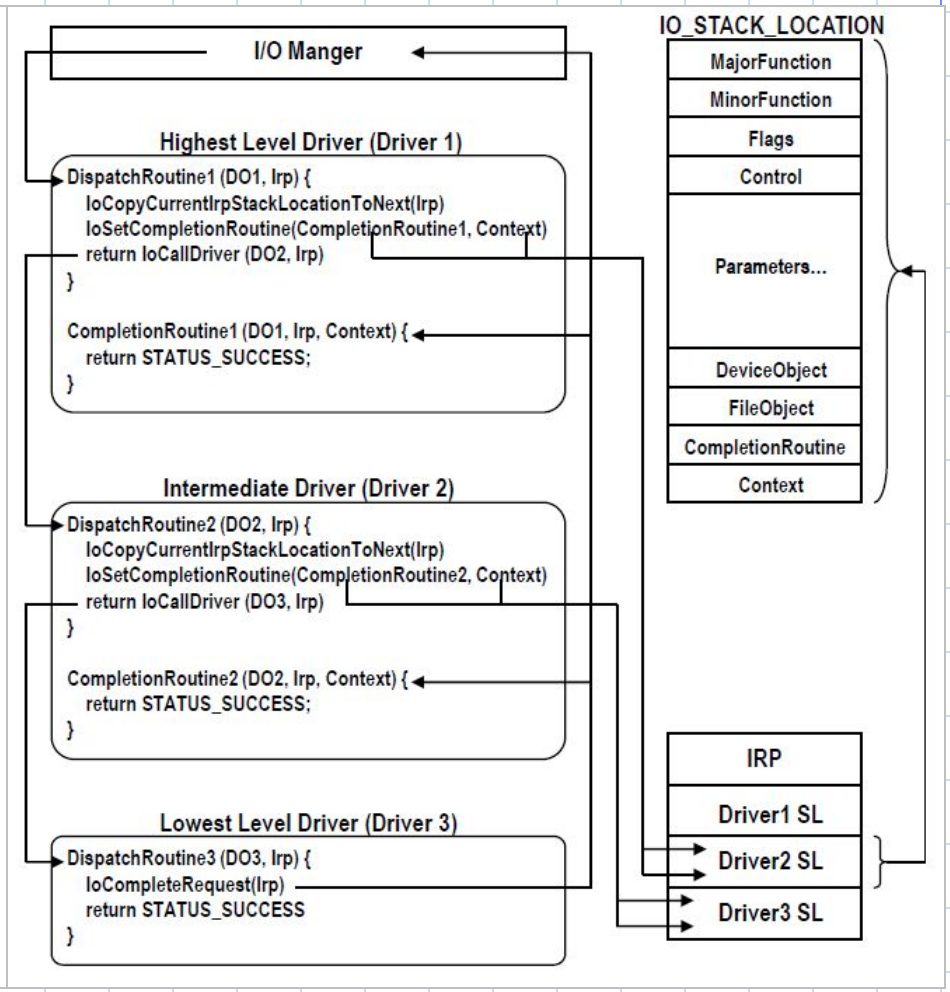


Схема обработки IRP

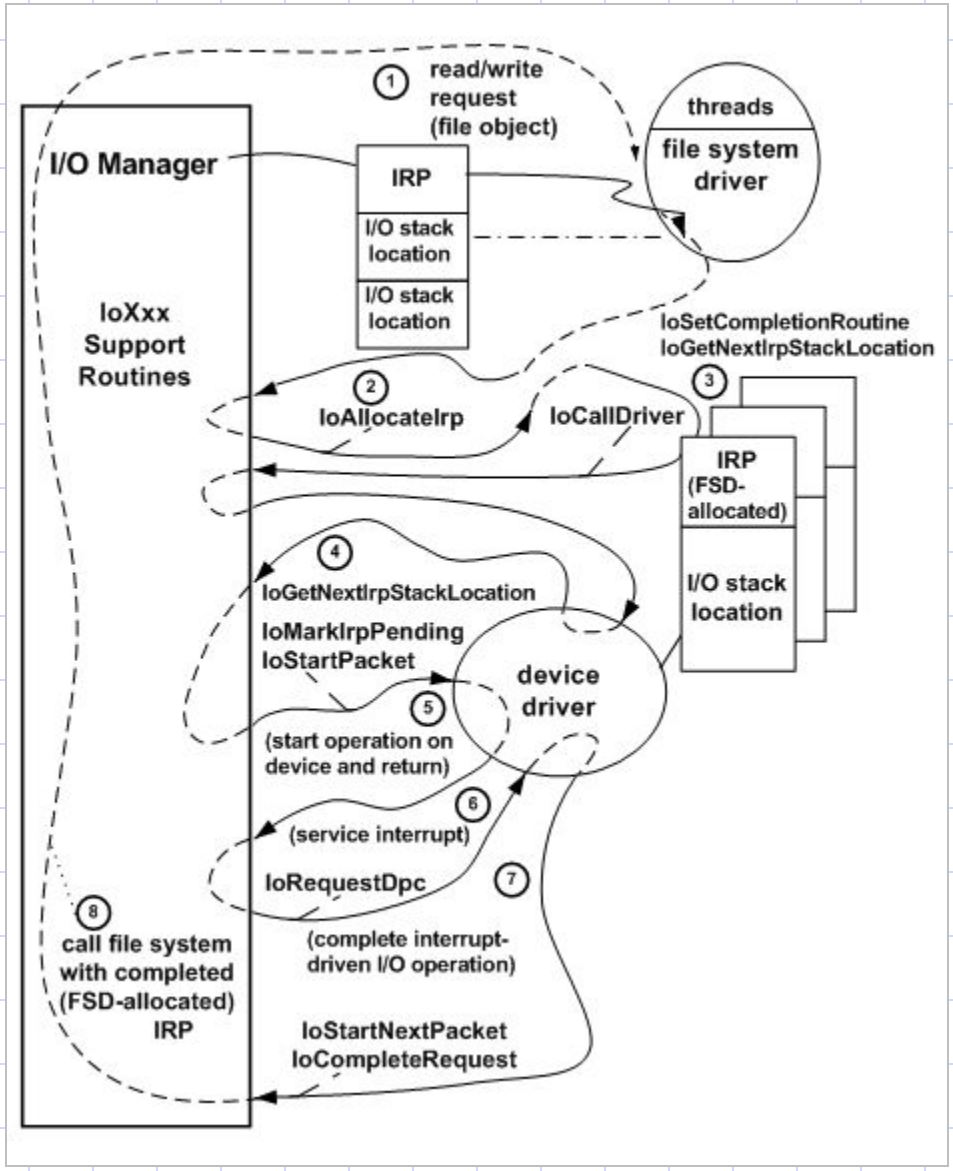




Алгоритм обработки IRP при открытии файла

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.

Детализированная схема обработки IRP



Алгоритм обработки IRP при операции с файлом

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.

Обработка IRP-пакетов

- Статьи в MSDN:
  - http://msdn.microsoft.com/en-us/library/windows/hardware/ff546847(v=vs.85).aspx
  - MSDN-ProcessingIRPOverview.mht
  - MSDN-ProcessingIRPDetails.mht

Перехват API-вызовов в user mode

- 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.

Перехват API-вызовов в user mode

- Внедрение DLL с помощью ловушек:
  - HHOOK **SetWindowsHookEx**(int idHook, HOOKPROC lpfn, INSTANCE hMod, DWORD dwThreadId); **UnhookWindowsHookEx**().
  - idHook: WH_CALLWNDPROC, WH_CALLWNDPROCRET, WH_CBT, WH_DEBUG, WH_FOREGROUNDIDLE, WH_GETMESSAGE, WH_JOURNALPLAYBACK, WH_JOURNALRECORD, WH_KEYBOARD, WH_KEYBOARD_LL, WH_MOUSE, WH_MOUSE_LL, WH_MSGFILTER, WH_SHELL, WH_SYSMSGFILTER.
- 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);

Перехват API-вызовов в user mode

- 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);

Перехват API-вызовов в user mode

- 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
  - См. в книге Джеффри Рихтера, глава 22.
- Перехват в точке входа в процедуру с помощью подмены начальных инструкций:
  - Библиотека Microsoft Detours: http://research.microsoft.com/en-us/projects/detours/
  - Detours: Binary Interception of Win32 Functions

Перехват API-вызовов в kernel mode

- 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.

Перехват API-вызовов в kernel mode

- 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.

Перехват API-вызовов в kernel mode

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

Object Manager Callbacks

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

Object Manager Callbacks

- Программирование процедур перехвата:
  - В callback-процедурах не делается различия между созданием объекта и созданием описателя, т.е. это создание описателя.
  - Процедуры вызываются на уровне приоритета прерываний PASSIVE_LEVEL в контексте потока, вызывающего создание описателя.
  - Процедура **ObjectPreCallback**() вызывается перед созданием описателя. Она принимает в параметрах оригинальную битовую маску прав, запрошенных для объекта пользователем (OperationInfo-> Parameters->CreateHandleInformation->OriginalDesiredAccess), и может ее скорректировать (OperationInfo->Parameters->CreateHandleInformation-> DesiredAccess).
  - Процедура **ObjectPostCallback**() вызывается после создания описателя. Она принимает в параметрах битовую маску прав, с которыми описатель был создан (OperationInfo->Parameters-> CreateHandleInformation->GrantedAccess).

Process Callbacks

- 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.

Thread Callbacks

- 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.

Module Load Callbacks

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

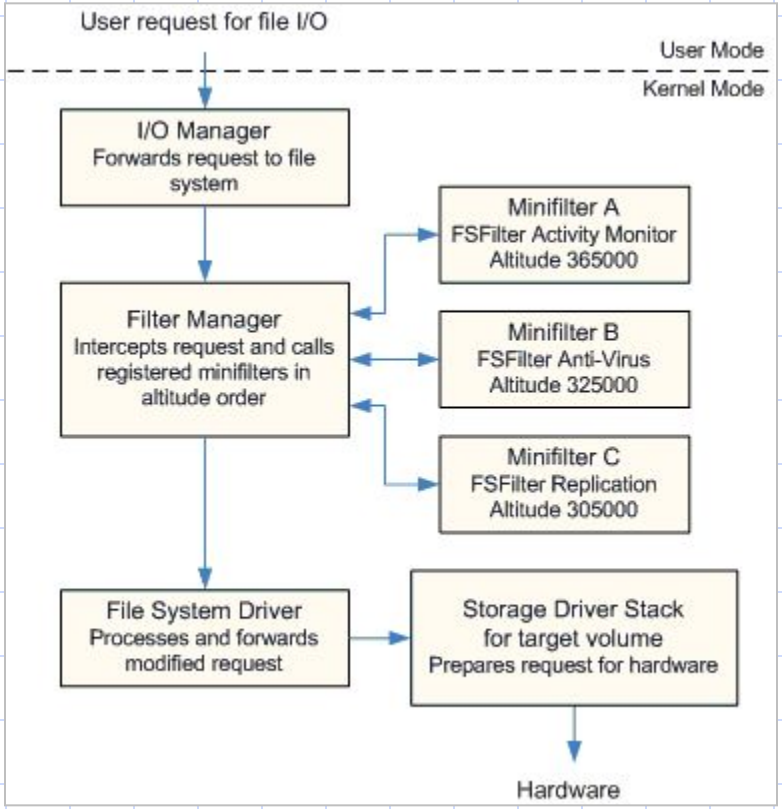
Registry Callbacks

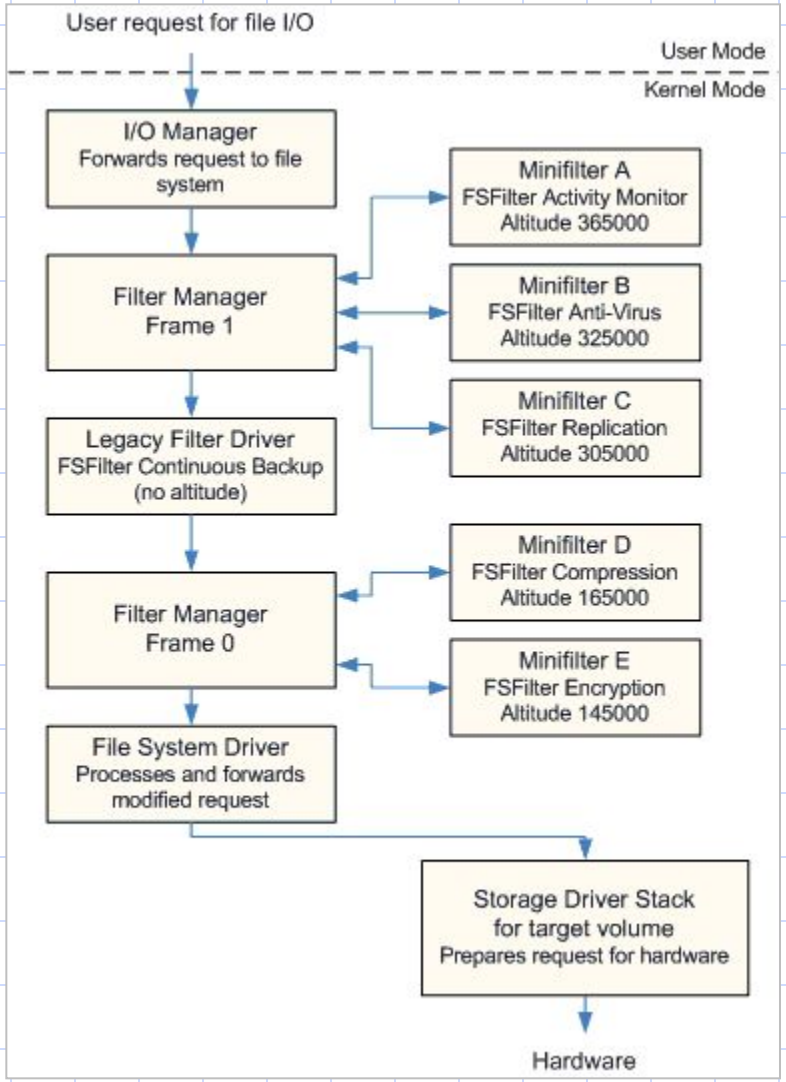
- Процедура перехвата операций с реестром Windows:
  - NTSTATUS **CmRegisterCallbackEx**(EX_CALLBACK_FUNCTION* Function, const UNICODE_STRING* Altitude, void* Driver, void* Context, LARGE_INTEGER* Cookie, void* Reserved); **CmRegisterCallback**() устарела.
  - NTSTATUS **CmUnRegisterCallback**(LARGE_INTEGER Cookie);
  - NTSTATUS **RegistryCallback**(void* Context, REG_NOTIFY_CLASS Argument1, void* Argument2); // Argument2 – указатель на REG_XXX_INFORMATION.
  - Процедура перехвата операций с реестром может выполнять: мониторинг, блокировку (XP и выше) и модификацию (Vista и выше).
  - Если операция предотвращается с помощью STATUS_CALLBACK_BYPASS, вызывающий поток получает STATUS_SUCCESS. Если операция предотвращается с помощью кода ошибки, поток получает ее код.
  - В процедуре перехвата при создании ключа реестра можно назначить ключу отдельный контекст, который будет передаваться в процедуру с уведомлениями REG_XXX_KEY_INFORMATION:
  - NTSTATUS **CmSetCallbackObjectContext**(void* Object, LARGE_INTEGER* Cookie, void* NewContext, void** OldContext);
  - При ассоциации контекста с ключом реестра, процедуре перехвата будет послано уведомление об уничтожении ключа.

File System Mini-Filter

- Перехват процедур взаимодействия программ с файловой системой возможен в двух вариантах:
  - Установка фильтрующего драйвера – File System Filter Driver. Устаревший способ, унаследованный от предыдущих версий Windows (до Win 2000).
  - Установка фильтрующего мини-драйвера (мини-фильтра) – File System Mini-Filter. Способ основан на запуске компонента Filter Manager (fltmgr.sys) как фильтрующего драйвера, запускающего и контролирующего мини-фильтры.
- Filter Manager:
  - Активизируется при загрузке первого же мини-фильтра. Обеспечивает работу множества мини-фильтров. Может загружать и выгружать мини-фильтры без перезагрузки системы.
  - Устанавливается в стеке драйверов между менеджером ввода-вывода и драйверами файловой системы (NTFS, FAT и др.). Умеет устанавливаться в обхват других фильтрующих драйверов (см. рисунок). При установке мини-фильтров применяет параметр высоты установки (Altitude).
  - Скрывает сложность модели ввода-вывода на основе IRP-пакетов и предоставляет интерфейс для регистрации функций перехвата. Параметры в функции перехвата приходят разобранными (не IRP).
  - Предоставляет API и для ядра, и для пользовательского режима.

File System Mini-Filter





http://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx

File System Mini-Filter

- Mini-Filter:
  - Загружается и управляется как из kernel mode, так и из user mode. Функции режима ядра – **FltXxxYyy**(), функции приложений – **FilterXxx**().
  - Перехватчики устанавливаются мини-фильтром в режиме ядра.
  - Для каждого тома файловой системы и присоединенного к нему мини-фильтра Filter Manager создает ассоциированный объект, называемый Instance. Мини-фильтр может перехватывать создание и удаление таких объектов.
- Загрузка и выгрузка мини-фильтра:
  - NTSTATUS **FltLoadFilter**(PCUNICODE_STRING FilterName);
  - NTSTATUS **FltUnloadFilter**(PCUNICODE_STRING FilterName);
  - HRESULT **FilterLoad**(LPCWSTR lpFilterName); // user mode
  - HRESULT **FilterUnload**(LPCWSTR lpFilterName); // user mode
- Регистрация процедур перехвата в мини-фильтре:
  - NTSTATUS **FltRegisterFilter**(DRIVER_OBJECT* Driver, const FLT_REGISTRATION* Registration, PFLT_FILTER* RetFilter);
  - NTSTATUS **FltStartFiltering**(PFLT_FILTER Filter);
  - void **FltUnregisterFilter**(PFLT_FILTER Filter);

File System Mini-Filter

- Регистрация процедур перехвата в мини-фильтре:

```
struct FLT_REGISTRATION
{
    USHORT Size; USHORT Version; FLT_REGISTRATION_FLAGS Flags;
    const FLT_CONTEXT_REGISTRATION* ContextRegistration;
    const FLT_OPERATION_REGISTRATION* OperationRegistration;
    FLT_FILTER_UNLOAD_CALLBACK* FilterUnloadCallback;
    FLT_INSTANCE_SETUP_CALLBACK* InstanceSetupCallback;
    FLT_INSTANCE_QUERY_TEARDOWN_CALLBACK* InstanceQueryTeardownCallback;
    FLT_INSTANCE_TEARDOWN_CALLBACK* InstanceTeardownStartCallback;
    FLT_INSTANCE_TEARDOWN_CALLBACK* InstanceTeardownCompleteCallback;
    FLT_GENERATE_FILE_NAME* GenerateFileNameCallback;
    FLT_NORMALIZE_NAME_COMPONENT* NormalizeNameComponentCallback;
    FLT_NORMALIZE_CONTEXT_CLEANUP* NormalizeContextCleanupCallback;
    FLT_TRANSACTION_NOTIFICATION_CALLBACK* TransactionNotificationCallback;
    FLT_NORMALIZE_NAME_COMPONENT_EX* NormalizeNameComponentExCallback;
```

File System Mini-Filter

- Контексты Filter Manager-а в мини-фильтре:
  - Если мини-фильтру нужно ассоциировать свои данные с объектами Filter Manager-а, он устанавливает массив перехватчиков на каждый из таких объектов – так называемых контекстов.
  - Существуют следующие контексты: том (volume), экземпляр мини-фильтра для тома (instance), поток ввода-вывода (stream), описатель потока ввода-вывода (stream handle), секция отображаемого файла (section), транзакция (transaction), файл (file). См. поле ContextType ниже.
  - struct FLT_CONTEXT_REGISTRATION
    {
    FLT_CONTEXT_TYPE ContextType; // обязательное поле
    FLT_CONTEXT_REGISTRATION_FLAGS Flags;
    FLT_CONTEXT_CLEANUP_CALLBACK* ContextCleanupCallback;
    SIZE_T Size; // размер порции мини-фильтра в общем контексте FM
    ULONG PoolTag;
    FLT_CONTEXT_ALLOCATE_CALLBACK* ContextAllocateCallback;
    FLT_CONTEXT_FREE_CALLBACK* ContextFreeCallback;
    void* Reserved1;
    };

File System Mini-Filter

- Регистрация процедур перехвата операций ввода-вывода в мини-фильтре:
  - struct FLT_OPERATION_REGISTRATION
    {
    UCHAR MajorFunction; // обязательное поле
    FLT_OPERATION_REGISTRATION_FLAGS Flags;
    FLT_PRE_OPERATION_CALLBACK* PreOperation;
    FLT_POST_OPERATION_CALLBACK* PostOperation;
    void* Reserved1;
    };
  - FLT_PREOP_CALLBACK_STATUS **PreOperationCallback**(
    FLT_CALLBACK_DATA* Data, const FLT_RELATED_OBJECTS* FltObjects,
    PVOID* CompletionContext);
  - FLT_POSTOP_CALLBACK_STATUS **PostOperationCallback**(
    FLT_CALLBACK_DATA* Data, const FLT_RELATED_OBJECTS* FltObjects,
    PVOID CompletionContext, FLT_POST_OPERATION_FLAGS Flags);
  - struct FLT_RELATED_OBJECTS { const USHORT Size;
    const USHORT TransactionContext; const FLT_FILTER* Filter;
    const FLT_VOLUME* Volume; const FLT_INSTANCE* Instance;
    const FILE_OBJECT* FileObject; const KTRANSACTION* Transaction; };