Анимация

Лекция 5

Программа «Прыгающий мячик»

- В этой программе демонстрируется применение следующих средств Win32 API:
- Использование узорной кисти (CreatePatternBrush) для фона окна (SetClassLong) и для операции стирания (FillRect) предыдущего изображения мяча.
- Контекст устройства в памяти (CreateCompatibleDC) для размещения в нем DDB- растра с изображением мяча (SelectObject) и последующего его вывода в контекст дисплея (BitBlt).
- Использование региона отсечения (CreateELlipticRgn, SelectClipRgn) для выделения в прямоугольном растре области с изображением мяча, которая затем копируется при помощи функции BitBlt.
- Мировые преобразования (SetWorldTransform) для перемещения и вращения изображения мяча.
- Функции SaveDC и RestoreDC, применяемые для сохранения и восстановления текущего состояния контекста устройства.

```
case WM_CREATE:
    hDC = GetDC(hWnd);
    GetClientRect(hWnd, &rect);
    dX = rect.right / 100.;
    dY = rect.bottom / 50.;
        // Создать таймер (0.1 сек)
    SetTimer(hWnd, 1, 100, NULL);

    hBmpBkgr = LoadBitmap((HINSTANCE)GetWindowLong(hWnd,
    GWL_HINSTANCE),MAKEINTRESOURCE(IDB_STONE));

    hBkBrush = CreatePatternBrush(hBmpBkgr);
    SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBkBrush);

    hBmpBall = LoadBitmap((HINSTANCE)GetWindowLong(hWnd,
    GWL_HINSTANCE),MAKEINTRESOURCE(IDB_BALL));

    GetObject(hBmpBall, sizeof(bm), (LPSTR)&bm);
    SetGraphicsMode(hDC, GM_ADVANCED);
    break;
```

```
case WM_TIMER:
    GetClientRect(hWnd, &rect);
    // Стираем прежнюю картинку мяча
    SetRect(&rBall, (int)x, (int)y, (int)x + bm.bmWidth, (int)y + bm.bmHeight);
    FillRect(hDC, &rBall, hBkBrush);

    // Новая позиция мяча
    x += dX;
    y += dY;
    alpha += 10;
    if (alpha > 360)  alpha = 0;

    // Если мяч достиг края окна, направление его движения изменяется
    if(x + bm.bmWidth > rect.right || x < 0)
    dX = -dX;
    if(y + bm.bmHeight > rect.bottom || y < 0)
    dY = -dY;
    DrawBall(hWnd, hDC, hBmpBall, bm, x, y, alpha);
    break;
```

```
case WM_DESTROY:
    KillTimer(hWnd, 1);
    ReleaseDC(hWnd, hDC);
    PostQuitMessage(0);
    break;
```

```
void DrawBall(HWND hwnd, HDC hdc, HBITMAP hBmp,
    BITMAP bm, FLOAT x, FLOAT y, int alpha) {
    XFORM xform;
    HRGN hRgn;

    // Подготовка к выводу мяча
    HDC hBallMemDC = CreateCompatibleDC(hdc);
    SelectObject(hBallMemDC, hBmp);

    // Создаем регион отсечения
    hRgn = CreateEllipticRgn(x, y, x + bm.bmWidth, y +
    bm.bmHeight);
    SelectClipRgn(hdc, hRgn);
```

```
// Мировые преобразования для перемещения и
   //вращения мяча
   xform.eM11 = (FLOAT) cos(alpha * 2 * Pi / 360);
   //вращение
   xform.eM12 = (FLOAT) sin(alpha * 2 * Pi / 360);
   //вращение
   xform.eM21 = (FLOAT) -sin(alpha * 2 * Pi / 360);
   //вращение
   xform.eM22 = (FLOAT) cos(alpha * 2 * Pi / 360);
   //вращение
   xform.eDx  = x + bm.bmWidth / 2.;
   //смещение по оси x
   xform.eDy  = y + bm.bmHeight / 2.;
   //смещение по оси y
```

```
// Вывод мяча
    SaveDC(hdc);
    BOOL ret = SetWorldTransform(hdc, &xform);
    BitBlt(hdc, -bm.bmWidth/2, -bm.bmHeight/2,
                    bm.bmWidth, bm.bmHeight, hBallMemDC,
    0, 0, SRCCOPY);
    RestoreDC(hdc, -1);

    SelectClipRgn(hdc, NULL);
    DeleteObject(hRgn);
    DeleteDC(hBallMemDC);
```

Полный текст файла

- лекция 5 код1.docx

Вызов функции SetGraphicsMode для переключения контекста устройства в графический режим (GM_ADVANCED).

Этот режим нужен для использования мировой системы координат и мировых преобразований. В теле функции DrawBall мировые преобразования реализуются вызовом функции SetWorld-Transform, а изменяющиеся значения полей структуры xform обеспечивают эффект перемещения и вращения мяча. Эти преобразования должны использоваться *только при выводе изображения мяча.*

Поэтому перед вызовом SetWorldTransform мы запоминаем текущее состояние контекста устройства с помощью Save DC, а затем восстанавливаем его вызовом RestoreDC.

Основные события разворачиваются в блоке обработки сообщения WM_TIMER.

- Все работает замечательно, за исключением одной детали. Дело в том, что вращающийся мячик мерцает.
- Причиной этого является быстрое последовательное выполнение двух операций с контекстом дисплея: стирание прежнего изображения мяча (FillRect) и вывод нового изображения (BitBlt).
- Вы можете убедиться в этом, закомментировав вызов функции FillRect. После этого изображение перестанет мерцать, но вместо летающего мячика получится что-то вроде червя, прогрызающего тоннель в камне.

Двойная буферизация

- Неприятное мерцание изображения в анимационном приложении можно устранить, если сформировать очередную фазу картинки в *виртуальном контексте устройства.*
- Для этого используется контекст в памяти, совместимый с контекстом дисплея.
- В нашем случае очередная фаза содержит две операции:
- а) стереть предшествующее изображение мяча;
- б) нарисовать новое изображение мяча.
- После этого содержимое совместимого контекста копируется в контекст дисплея.

Двойная буферизация

- Двойная буферизация в программе реализована на основе контекста в памяти hMemDcFrame, совместимого с контекстом дисплея.
- Виртуальный контекст создается вызовом функции CreateCompatibleDC, после чего в него выбирается при помощи функции SelectObject растр hBmpFrame, также совместимый с контекстом дисплея и имеющий размеры клиентской области окна приложения.
- Инициализация контекста hMemDcFrame для копирования в него изображения фона осуществляется в блоке обработки сообщения WM_TIMER.
- Чтобы инициализация была однократной, мы используем счетчик count и вызываем функцию BitBlt только при нулевом значении счетчика.

- Также нужно позаботиться о новой инициализации контекста hMemDcFrame в случае изменения размеров окна. Для этого добавлен код обработки сообщения WM_SIZE.
- Стирание прежней картинки мяча в виртуальном контексте происходит в блоке обработки сообщения WM_TIMER.
- После вычисления новых координат мяча вызывается функция DrawBall, и ей передается виртуальный контекст в качестве параметра hMemFrameDC.
- В теле функции DrawBall завершается формирование очередной фазы картинки, когда вызывается функция BitBlt для вывода изображения мяча в виртуальный контекст.
- Только после этого вся картинка копируется при помощи второго вызова BitBlt из hMemFrameDC в контекст дисплея.

Полный код лекция 5 код2.docx

```
case WM_CREATE:
    hDC = GetDC(hWnd);
    GetClientRect(hWnd, &rect);
    dX = rect.right / 100.;
    dY = rect.bottom / 50.;

    // Создать таймер (0.1 сек)
    SetTimer(hWnd, 1, 100, NULL);

    hBmpBkgr = LoadBitmap((HINSTANCE)GetWindowLong(hWnd,
          GWL_HINSTANCE), MAKEINTRESOURCE(IDB_STONE));
    hBkBrush = CreatePatternBrush(hBmpBkgr);
    SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBkBrush);
```

```
hBmpBall = LoadBitmap((HINSTANCE)GetWindowLong(hWnd,
        GWL_HINSTANCE),  MAKEINTRESOURCE(IDB_BALL));
GetObject(hBmpBall, sizeof(bm), (LPSTR)&bm);
    hMemDcFrame = CreateCompatibleDC(hDC);
    hBmpFrame = CreateCompatibleBitmap(hDC, rect.right,
    rect.bottom);
    SelectObject(hMemDcFrame, hBmpFrame);
    SetGraphicsMode(hMemDcFrame, GM_ADVANCED);
break;


case WM_SIZE:
    GetClientRect(hWnd, &rect);
    hBmpFrame = CreateCompatibleBitmap(hDC, rect.right,
    rect.bottom);
    DeleteObject(SelectObject(hMemDcFrame, hBmpFrame));
    // Копирование фона в hMemDcFrame
    BitBlt(hMemDcFrame, 0, 0, rect.right, rect.bottom, hDC, 0, 0,
    SRCCOPY);
    break;
```

```
case WM_TIMER:
    GetClientRect(hWnd, &rect);
    if (!count)
{    // Копирование фона в hMemDcFrame
    BitBlt(hMemDcFrame, 0, 0, rect.right, rect.bottom, hDC, 0,
    0, SRCCOPY);
    count++;
}
    // Стираем прежнюю картинку мяча
    SetRect(&rBall, x, y, x + bm.bmWidth, y + bm.bmHeight);
    FillRect(hMemDcFrame, &rBall, hBkBrush);
```

```
// Новая позиция мяча
    x += dX;
    y += dY;
    alpha += 10;
    if (alpha > 360)  alpha = 0;

        // Если мяч достиг края окна, направление его //движения
    изменяется
    if(x + bm.bmWidth > rect.right || x < 0)          dX = -dX;
    if(y + bm.bmHeight > rect.bottom || y < 0)     dY = -dY;

    DrawBall(hWnd, hDC, hMemDcFrame, hBmpBall, bm, (int)x, (int)y, alpha);
    break;

case WM_DESTROY:
    KillTimer(hWnd, 1);
    ReleaseDC(hWnd, hDC);
    DeleteDC(hMemDcFrame);
    PostQuitMessage(0);
    break;
```

```
void DrawBall(HWND hwnd, HDC hdc, HDC hMemFrameDC,
    HBITMAP hBmp,  BITMAP bm, FLOAT x, FLOAT y, int alpha) {
    XFORM xform;
    HRGN hRgn;

    // Подготовка к выводу мяча
    HDC hMemDcBall = CreateCompatibleDC(hdc);
    SelectObject(hMemDcBall, hBmp);

    // Создаем регион отсечения
    hRgn = CreateEllipticRgn(x, y, x + bm.bmWidth, y +
    bm.bmHeight);
    SelectClipRgn(hMemFrameDC, hRgn);
```

```
// Мировые преобразования для перемещения и вращения мяча
    xform.eM11 = (FLOAT) cos(alpha * 2 * Pi / 360);  //вращение
    xform.eM12 = (FLOAT) sin(alpha * 2 * Pi / 360);  //вращение
    xform.eM21 = (FLOAT) -sin(alpha * 2 * Pi / 360); //вращение
    xform.eM22 = (FLOAT) cos(alpha * 2 * Pi / 360);  //вращение
    xform.eDx  = x + bm.bmWidth / 2;       //смещение по оси x
    xform.eDy  = y + bm.bmHeight / 2;      //смещение по оси y

    // Вывод мяча в контекст hMemFrameDC
    SaveDC(hMemFrameDC);
    BOOL ret = SetWorldTransform(hMemFrameDC, &xform);
    BitBlt(hMemFrameDC, -bm.bmWidth/2, -bm.bmHeight/2,
           bm.bmWidth, bm.bmHeight, hMemDcBall, 0, 0, SRCCOPY);
    RestoreDC(hMemFrameDC, -1);
```

```
// Копирование изображения из hMemFrameDC в hdc
    RECT rect;
    GetClientRect(hwnd, &rect);
    BitBlt(hdc, 0, 0, rect.right, rect.bottom,
    hMemFrameDC, 0, 0, SRCCOPY);

    SelectClipRgn(hMemFrameDC, NULL);
    DeleteObject(hRgn);
    DeleteDC(hMemDcBall);
}
```

Хуки

- В операционной системе Windows хуком называется механизм перехвата особой функцией событий (таких как сообщения, ввод с мыши или клавиатуры) до того, как они дойдут до приложения. Эта функция может затем реагировать на события и, в некоторых случаях, изменять или отменять их.
- Функции, получающие уведомления о событиях, называются *фильтрующими функциями* и различаются по типам перехватываемых ими событий.
- Пример - фильтрующая функция для перехвата всех событий мыши или клавиатуры. Чтобы Windows смогла вызывать функцию-фильтр, эта функция должна быть установлена - то есть, прикреплена - к хуку (например, к клавиатурному хуку).
- Прикрепление одной или нескольких фильтрующих функций к какому-нибудь хуку называется *установкой* хука.

- Если к одному хуку прикреплено несколько фильтрующих функций, Windows реализует очередь функций, причем функция, прикрепленная последней, оказывается в начале очереди, а самая первая функция - в ее конце.
- Когда к хуку прикреплена одна или более функций-фильтров и происходит событие, приводящее к срабатыванию хука, Windows вызывает первую функцию из очереди функций-фильтров.
- Это действие называется вызовом хука. К примеру, если к хуку CBT прикреплена функция и происходит событие, после которого срабатывает хук (допустим, идет создание окна), Windows вызывает CBT-хук, то есть первую функцию из его очереди.
- Для установки и доступа к фильтрующим функциям приложения используют функции **SetWindowsHookEx** и **UnhookWindowsHookEx**.

Приложения могут использовать хуки в следующих целях:

- Обрабатывать или изменять все сообщения, предназначенные для всех диалоговых окон (dialog box), информационных окон (message box), полос прокрутки (scroll bar), или меню одного приложения (WH_MSGFILTER).
- Обрабатывать или изменять все сообщения, предназначенные для всех диалоговых окон, информационных окон, полос прокрутки, или меню всей системы (WH_SYSMSGFILTER).
- Обрабатывать или изменять все сообщения в системе (все виды сообщений), получаемые функциями **GetMessage** или **PeekMessage** (WH_GETMESSAGE).
- Обрабатывать или изменять все сообщения (любого типа), посылаемые вызовом функции **SendMessage** (WH_CALLWNDPROC).

Использование хуков

- Записывать или проигрывать клавиатурные и мышиные события (WH_JOURNALRECORD, WH_JOURNALPLAYBACK).
- Обрабатывать, изменять или удалять клавиатурные события (WH_KEYBOARD).
- Обрабатывать, изменять или отменять события мыши (WH_MOUSE).
- Реагировать на определенные действия системы, делая возможным разработку приложений компьютерного обучения - computer-based training (WH_CBT).
- Предотвратить вызов другой функции-фильтра (WH_DEBUG).

Примеры использования хуков

- Добавить поддержку кнопки F1 для меню, диалоговых и информационных окон (WH_MSGFILTER).
- Обеспечить запись и воспроизведение событий мыши и клавиатуры, часто называемых макросами. Например, программа Windows Recorder использует хуки для записи и воспроизведения (WH_JOURNALRECORD, WH_JOURNALPLAYBACK).
- Следить за сообщениями, чтобы определить, какие сообщения предназначены определенному окну или какие действия генерирует сообщение (WH_GETMESSAGE, WH_CALLWNDPROC). Утилита Spy из Win32™ Software Development Kit (SDK) for Windows NT™ использует для этих целей хуки. Исходные тексты Spy можно найти в SDK.
- Симулировать мышиный и клавиатурный ввод (WH_JOURNALPLAYBACK). Хуки - единственный надежный способ симуляции этих действий. Если попытаться имитировать их через посылку сообщений, не будет происходить обновление состояния клавиатуры или мыши во внутренних структурах Windows, что может привести к непредсказуемому поведению. Если для воспроизведения клавиатурных или мышиных событий используются хуки, эти события обрабатываются в точности так, как и настоящий ввод с клавиатуры или мыши. Microsoft Excel использует хуки для реализации макрофункции SEND.KEYS.

Как пользоваться хуками

Чтобы пользоваться хуками, вам необходимо знать следующее:

- Как использовать функции Windows для добавления и удаления фильтрующих функций из очереди функций хука.
- Какие действия должна будет выполнить фильтрующая функция, которую вы устанавливаете.
- Какие существуют виды хуков, что они могут делать, и какую информацию (параметры) они передают вашей функции.

Функции Windows для работы с хуками

Приложения Windows используют функции

- **SetWindowsHookEx**
- **UnhookWindowsHookEx**
- **CallNextHookEx**

для управления очередью функций-фильтров хука.

SetWindowsHookEx

- Функция **SetWindowsHookEx** добавляет функцию-фильтр к хуку. Эта функция принимает четыре аргумента:
- Целочисленный код, описывающий хук, к которому будет прикреплена фильтрующая функция. Эти коды определены в WINUSER.H.
- Адрес функции-фильтра. Эта функция должна быть описана как экспортируемая включением ее в секцию **EXPORTS** файла определения приложения или библиотеки динамической линковки (DLL), или использованием соответствующих опций компилятора.
- Хэндл модуля, содержащего фильтрующую функцию. В Win32 , этот параметр должен быть NULL при установке хука на поток, но данное требование не является строго обязательным, как указано в документации. При установке хука для всей системы или для потока в другом процессе, нужно использовать хэндл DLL, содержащей функцию-фильтр.
- Идентификатор потока, для которого устанавливается хук. Если этот идентификатор ненулевой, установленная фильтрующая функция будет вызываться только в контексте указанного потока. Если идентификатор равен нулю, установленная функция имеет системную область видимости и может быть вызвана в контексте любого потока в системе. Приложение или библиотека могут использовать функцию GetCurrentThreadId для получения идентификатора текущего потока.

Область видимости хука

Хук	Область видимости
WH_CALLWNDPROC	Поток или вся система
WH_CBT	Поток или вся система
WH_DEBUG	Поток или вся система
WH_GETMESSAGE	Поток или вся система
WH_JOURNALRECORD	Только система
WH_JOURNALPLAYBACK	Только система
WH_FOREGROUNDIDLE	Поток или вся система
WH_SHELL	Поток или вся система
WH_KEYBOARD	Поток или вся система
WH_MOUSE	Поток или вся система
WH_MSGFILTER	Поток или вся система
WH_SYSMSGFILTER	Только система

Для любого данного типа хука, первыми вызываются хуки потоков, и только затем системные хуки.

Хуки потоков:

- Не создают лишней работы приложениям, которые не заинтересованы в вызове хука.
- Не помещают все события, относящиеся к хуку, в очередь (так, чтобы они поступали не одновременно, а одно за другим). Например, если приложение установит клавиатурный хук для всей системы, то все клавиатурные сообщения будут пропущены через фильтрующую функцию этого хука, оставляя неиспользованными системные возможности многопотоковой обработки ввода. Если эта функция прекратит обрабатывать клавиатурные события, система будет выглядеть зависшей, хотя на самом деле и не зависнет.
- Не требуют нахождения функции-фильтра в отдельной DLL. Все системные хуки и хуки для потоков в другом приложении должны находиться в DLL.

- **SetWindowsHookEx** возвращает хэндл установленного хука (тип HHOOK).
- Приложение или библиотека должны использовать этот хэндл для вызова функции **UnhookWindowsHookEx**.
- **SetWindowsHookEx** возвращает NULL если она не смогла добавить функцию к хуку.
- **SetWindowsHookEx** также устанавливает код последней ошибки в одно из следующих значений для индикации неудачного завершения функции.
- Windows сама заботится об организации очереди функций-фильтров не доверяя функциям хранение адресов следующих функций в очереди.

- **ERROR_INVALID_HOOK_FILTER**: Неверный код хука.
- **ERROR_INVALID_FILTER_PROC**: Неверная фильтрующая функция.
- **ERROR_HOOK_NEEDS_HMOD**: Глобальный хук устанавливается с параметром *hInstance*, равным NULL либо локальный хук устанавливается для потока, который не принадлежит данному приложению.
- **ERROR_GLOBAL_ONLY_HOOK**: Хук, который может быть только системным, устанавливается как потоковый.
- **ERROR_INVALID_PARAMETER**: Неверный идентификатор потока.
- **ERROR_JOURNAL_HOOK_SET:** Для регистрационного хука (journal hook) уже установлена фильтрующая функция. В любой момент времени может быть установлен только один записывающий или воспроизводящий хук. Этот код ошибки может также означать, что приложение пытается установить регистрационный хук в то время, как запущен хранитель экрана.
- **ERROR_MOD_NOT_FOUND:** Параметр *hInstance* в случае, когда хук является глобальным, не ссылался на библиотеку. (На самом деле, это значение означает лишь, что модуль User не смог обнаружить данный хэндл в списке модулей.)
- Любое другое значение: Система безопасности не позволяет установить данный хук, либо в системе закончилась память.

UnhookWindowsHookEx

- Для удаления функции-фильтра из очереди хука вызовите функцию **UnhookWindowsHookEx**.
- Эта функция принимает хэндл хука, полученный от **SetWindowsHookEx** и возвращает логическое значение, показывающее успех операции.
- На данный момент **UnhookWindowsHookEx** всегда возвращает TRUE.

Фильтрующие функции

- Фильтрующие (*хуковые*) функции - это функции, прикрепленные к хуку. Из-за того, что эти функции вызываются Windows, а не приложением, их часто называют *функциями обратного вызова* (callback functions).
- Все фильтрующие функции должны быть описаны следующим образом:

- LRESULT CALLBACK FilterFunc(int nCode, WPARAM wParam, LPARAM lParam)

- Все функции-фильтры должны возвращать **LONG**. Вместо *FilterFunc* должно стоять имя вашей фильтрующей функции.

Параметры

- Фильтрующие функции принимают три параметра: *nCode* (код хука), *wParam*, и *lParam*. Код хука - это целое значение, которое передает функции дополнительную информацию. К примеру, код хука может описывать событие, которое привело к срабатыванию хука.
- Второй параметр функции-фильтра, *wParam*, имеет тип WPARAM, и третий параметр, *lParam*, имеет тип LPARAM. Эти параметры передают информацию фильтрующим функциям.
- У каждого хука значения *wParam* и *lParam* различаются. Например, фильтры хука WH_KEYBOARD получают в *wParam* виртуальный код клавиши, а в *lParam* - состояние клавиатуры на момент нажатия клавиши.
- Фильтрующие функции, прикрепленные к хуку WH_MSGFILTER получают в wParam значение NULL, а в *lParam* - указатель на структуру, описывающую сообщение.
- За полным описанием значений аргументов каждого типа хука обратитесь к Win32 SDK.

Вызов следующей функции в цепочке фильтрующих функций

- Когда хук уже установлен, Windows вызывает первую функцию в очереди, и на этом ее ответственность заканчивается. После этого функция ответственна за то, чтобы вызвать следующую функцию в цепочке.
- В Windows имеется функция **CallNextHookEx** для вызова следующего фильтра в очереди фильтров. **CallNextHookEx** принимает четыре параметра.
- Первый параметр - это значение, возвращенное функцией **SetWindowsHookEx**. В настоящее время Windows игнорирует это значение, но в будущем это может измениться.
- Следующие три параметра - *nCode*, *wParam*, и *lParam* - Windows передает дальше по цепочке функций.
- Windows хранит в своих внутренних структурах цепочку фильтрующих функций и следит за тем, какая функция вызывается в настоящий момент.
- При вызове **CallNextHookEx** Windows определяет следующую функцию в очереди и вызывает ее.

- Иногда функции-фильтры могут не пожелать передать обработку события другим фильтрам в той же цепочке. В частности, когда хук позволяет функции отменить событие и функция решает так поступить, она не должна вызывать **CallNextHookEx**. Когда фильтрующая функция модифицирует сообщение, она может решить не передавать его остальным функциям, ожидающим в очереди.
- Из-за того, что фильтры никак не сортируются при помещении их в очередь, вы не можете быть уверены, где находится ваша функция в любой момент времени кроме момента установки, когда ваша функция помещается в самое начало очереди. В результате, вы никогда не можете точно знать, что каждое событие в системе дойдет до вашего фильтра. Фильтрующая функция перед вашей функцией в цепочке - то есть функция, которая была установлена позже вашей - может не передать вам обработку события.

Хук WH_CALLWNDPROC

- Windows вызывает этот хук при каждом вызове функции **SendMessage**. Фильтрующей функции передается код хука, показывающий, была ли произведена посылка сообщения из текущего потока, а также указатель на структуру с информацией о сообщении.
- Структура CWPSTRUCT описана следующим образом:

```
typedef struct tagCWPSTRUCT {
LPARAM lParam;
 WPARAM wParam;
 DWORD message;
 HWND hwnd; } CWPSTRUCT, *LPCWPSTRUCT;
```

•Фильтры могут обработать сообщение, но не могут изменять его. Сообщение затем отсылается той функции, которой и предназначалось. Этот хук использует значительное количество системных ресурсов, особенно, когда он установлен с системной областью видимости, поэтому используйте его только в целях отладки.

Хук WH_CBT

- Чтобы написать приложение для интерактивного обучения (CBT application), разработчик должен координировать его работу с работой приложения, для которого оно разрабатывается. Для достижения этой цели Windows предоставляет разработчикам хук WH_CBT. Windows передает фильтрующей функции код хука, показывающий, какое произошло событие, и соответствующие этому событию данные.
- Фильтр для хука WH_CBT должен знать о десяти хуковых кодах:

- HCBT_ACTIVATE
- HCBT_CREATEWND
- HCBT_DESTROYWND
- HCBT_MINMAX
- HCBT_MOVESIZE
- HCBT_SYSCOMMAND
- HCBT_CLICKSKIPPED
- HCBT_KEYSKIPPED
- HCBT_SETFOCUS
- HCBT_QS

Код HCBT_ACTIVATE

- Windows вызывает хук WH_CBT с этим кодом при активации какого-нибудь окна. Когда хук WH_CBT установлен как локальный, это окно должно принадлежать потоку, на который установлен хук. Если фильтр в ответ на это событие вернет TRUE, окно не будет активизировано.
- Параметр *wParam* содержит хэндл активизируемого окна. В *lParam* содержится указатель на структуру **CBTACTIVATESTRUCT**, которая описана следующим образом:
- typedef struct tagCBTACTIVATESTRUCT {
- BOOL fMouse; // TRUE, если активация наступила в результате // мышиного клика; иначе FALSE.
- HWND hWndActive; // Содержит хэндл окна, активного // в настоящий момент.
- } CBTACTIVATESTRUCT, *LPCBTACTIVATESTRUCT;

Код HCBT_CREATEWND

- Windows вызывает хук WH_CBT с этим при создании окна. Когда хук установлен как локальный, это окно должно создаваться потоком, на который установлен хук. Хук WH_CBT вызывается до того, как Windows пошлет новому окну сообщения WM_GETMINMAXINFO, WM_NCCREATE, или WM_CREATE. Таким образом, фильтрующая функция может запретить создание окна, вернув TRUE.
- В параметре *wParam* содержится хэндл создаваемого окна. В *lParam* - указатель на следующую структуру.

- struct CBT_CREATEWND {
  struct tagCREATESTRUCT *lpcs; // Данные для создания нового окна.
  HWND hwndInsertAfter; // Хэндл окна, после которого будет
  // добавлено это окно (Z-order).
  } CBT_CREATEWND, *LPCBT_CREATEWND;
- Функция-фильтр может изменить значение *hwndInsertAfter* или значения в *lpcs*.

Код HCBT_DESTROYWND

- Windows вызывает хук WH_CBT с этим кодом перед уничтожением какого-либо окна. Если хук является локальным, это окно должно принадлежать потоку, на который установлен хук. Windows вызывает хук WH_CBT до посылки сообщения WM_DESTROY.
- Если функция-фильтр вернет TRUE, окно не будет уничтожено.
- Параметр *wParam* содержит хэндл уничтожаемого окна.
- В *lParam* находится 0L.

Код HCBT_MINMAX

- Windows вызывает хук WH_CBT с этим кодом перед минимизацией или максимизацией окна. Когда хук установлен как локальный, это окно должно принадлежать потоку, на который установлен хук. Если фильтр вернет TRUE, действие будет отменено.
- В *wParam* передается хэндл окна, которое готовится к максимизации/минимизации. *lParam* содержит одну из SW_*-констант, определенных в WINUSER.H и описывающих операцию над окном.

Код HCBT_MOVESIZE

- Windows вызывает хук WH_CBT с этим кодом перед перемещением или изменением размеров окна, сразу после того, как пользователь закончил выбор новой позиции или размеров окна. Если хук установлен как локальный, это окно должно принадлежать потоку, на который установлен хук. Если фильтр вернет TRUE, действие будет отменено.
- В *wParam* передается хэндл перемещаемого/изменяемого окна.
- *lParam* содержит **LPRECT**, который указывает на новые координаты окна.

Код HCBT_SYSCOMMAND

- Windows вызывает хук WH_CBT с этим кодом во время обработки системной команды. Если хук установлен как локальный, окно, чье системное меню вызвало данное событие, должно принадлежать потоку, на который установлен хук. Хук WH_CBT вызывается из функции **DefWindowsProc**. Если приложение не передает сообщение WH_SYSCOMMAND функции **DefWindowsProc**, это хук не получит управление. Если функция-фильтр вернет TRUE, системная команда не будет выполнена.
- В *wParam* содержится системная команда (SC_TASKLIST, SC_HOTKEY, и так далее), готовая к выполнению. Если в *wParam* передается SC_HOTKEY, в младшем слове (LOWORD) *lParam* содержится хэндл окна, к которому относится горячая клавиша. Если в *wParam* передается любое другое значение и если команда системного меню была выбрана мышью, в младшем слове *lParam* будет находиться горизонтальная позиция, а в старшем слове (HIWORD) - вертикальная позиция указателя мыши.
- Следующие системные команды приводят к срабатыванию этого хука изнутри **DefWindowProc**:

SC_CLOSE	Закрыть окно.
SC_HOTKEY	Активировать окно, связанное с определенной горячей клавишей.
SC_HSCROLL	Горизонтальная прокрутка.
SC_KEYMENU	Выполнить команду меню по комбинации клавиш.
SC_MAXIMIZE	Распахнуть окно.
SC_MINIMIZE	Минимизировать окно.
SC_MOUSEMENU	Выполнить команду меню по щелчку мыши.
SC_MOVE	Переместить окно.
SC_NEXTWINDOW	Перейти к следующему окну.
SC_PREVWINDOW	Перейти к предыдущему окну.
SC_RESTORE	Сохранить предыдущие координаты (контрольная точка - checkpoint).
SC_SCREENSAVE	Запустить хранитель экрана.
SC_SIZE	Изменить размер окна.
SC_TASKLIST	Запустить или активировать Планировщик Задач (Windows Task Manager).
SC_VSCROLL	Вертикальная прокрутка.

Код HCBT_CLICKSKIPPED

- Windows вызывает хук WH_CBT с этим кодом при удалении события от мыши из входной очереди потока, в случае, если установлен хук мыши. Windows вызовет системный хук, когда из какой-либо входной очереди будет удалено событие от мыши и в системе установлен либо глобальный, либо локальный хук мыши. Данный код передается только в том случае, если к хуку WH_MOUSE прикреплена фильтрующая функция. Несмотря на свое название, HCBT_CLICKSKIPPED генерируется не только для пропущенных событий от мыши, но и в случае, когда событие от мыши удаляется из системной очереди. Его главное назначение - установить хук WH_JOURNALPLAYBACK в ответ на событие мыши.
- В *wParam* передается идентификатор сообщения мыши - например, WM_LBUTTONDOWN или любое из сообщений WM_?BUTTON*. *lParam* содержит указатель на структуру **MOUSEHOOKSTRUCT**, которая описана следующим образом:
- typedef struct tagMOUSEHOOKSTRUCT {
- POINT pt; // Позиция курсора мыши в координатах экрана
- HWND hwnd; // Окно, получающее сообщение
- UINT wHitTestCode; // Результат проверки координат (hit-testing)
- DWORD dwExtraInfo; // Доп.информация о сообщении
- } MOUSEHOOKSTRUCT, *LPMOUSEHOOKSTRUCT;

Код HCBT_KEYSKIPPED

- Windows вызывает хук WH_CBT с этим кодом при удалении клавиатурного события из системной очереди, в случае, если установлен клавиатурный хук. Windows вызовет системный хук, когда из какой-либо входной очереди будет удалено событие от клавиатуры и в системе установлен либо глобальный, либо локальный клавиатурный хук.
- Данный код передается только в том случае, если к хуку WH_KEYBOARD прикреплена фильтрующая функция. Несмотря на свое название, HCBT_KEYSKIPPED генерируется не только для пропущенных клавиатурных событий, но и в случае, когда клавиатурное событие удаляется из системной очереди. Его главное назначение - установить хук WH_JOURNALPLAYBACK в ответ на клавиатурное событие.
- В *wParam* передается виртуальный код клавиши - то же самое значение, что и в *wParam* функций **GetMessage** или **PeekMessage** для сообщений WM_KEY*. *lParam* содержит то же значение, что и *lParam* функций **GetMessage** или **PeekMessage** для сообщений WM_KEY*.

WM_QUEUESYNC

- Часто приложение интерактивного обучения (Computer Based Training application или *CBT-приложение*) должно реагировать на события в процессе, для которого оно разработано. Обычно такими событиями являются события от клавиатуры или мыши. К примеру, пользователь нажимает на кнопку OK в диалоговом окне, после чего CBT-приложение желает послать главному приложению серию клавиатурных нажатий.
- CBT-приложение может использовать хук мыши для определения момента нажатия кнопки OK. После этого, CBT-приложение должно выждать некоторое время, пока главное приложение не закончит обработку нажатия кнопки OK (CBT-приложение вряд ли хочет послать клавиатурные нажатия диалоговому окну).
- CBT-приложение может использовать сообщение WM_QUEUESYNC для определения момента окончания нужного действия. Слежение производится с помощью клавиатурного или мышиного хуков. Наблюдая за главным приложением с помощью хуков, CBT-приложение узнает о наступлении необходимого события. После этого CBT-приложение должно подождать окончания этого события, прежде чем приступать к выполнению ответных действий.

Для определения момента окончания обработки события, CBT-приложение делает следующее:

- 1. Ждет от Windows вызова хука WH_CBT с кодом HCBT_CLICKSKIPPED или HCBT_KEYSKIPPED. Это происходит при удалении из системной очереди события, которое приводит к срабатыванию обработчика в главном приложении.
- 2. Устанавливает хук WH_JOURNALPLAYBACK. CBT-приложение не может установить этот хук, пока не получит код HCBT_CLICKSKIPPED или HCBT_KEYSKIPPED.

Хук WH_JOURNALPLAYBACK посылает CBT-приложению сообщение WM_QUEUESYNC. Когда CBT-приложение получает такое сообщение, оно может выполнить необходимые действия, например, послать главному приложению серию клавиатурных нажатий.

Код HCBT_SETFOCUS

- Windows вызывает хук WH_CBT с таким кодом, когда Windows собирается передать фокус ввода какому-либо окну. Когда хук установлен как локальный, это окно должно принадлежать потоку, на который установлен хук. Если фильтр вернет TRUE, фокус ввода не изменится.
- В *wParam* передается хэндл окна, получающего фокус ввода. *lParam* содержит хэндл окна, теряющего фокус ввода.

Код HCBT_QS

- Windows вызывает хук WH_CBT с этим кодом когда из системной очереди удаляется сообщение WM_QUEUESYNC, в то время как происходит изменение размеров или перемещение окна. Ни в каком другом случае этот хук не вызывается. Если хук установлен как локальный, это окно должно принадлежать потоку, на который установлен хук.
- Оба параметра - и *wParam*, и *lParam* - содержат ноль.

Хук WH_DEBUG

- Windows вызывает этот хук перед вызовом какой-либо фильтрующей функции. Фильтры не могут изменять значения, переданные этому хуку, но могут предотвратить вызов фильтрующей функции, возвратив ненулевое значение.
- В *wParam* передается идентификатор вызываемого хука, например, WH_MOUSE. *lParam* содержит указатель на следующую структуру:

```
typedef struct tagDEBUGHOOKINFO
{ DWORD idThread; // Идентификатор текущего потока
 LPARAM reserved;
 LPARAM lParam; // lParam для фильтрующей функции
 WPARAM wParam; // wParam для фильтрующей функции
 int code; } DEBUGHOOKINFO, * LPDEBUGHOOKINFO;
```

WH_FOREGROUNDIDLE

- Windows вызывает этот хук, когда к текущему потоку не поступает пользовательский ввод для обработки. Когда хук установлен как локальный, Windows вызывает его только при условии отсутствия пользовательского ввода у потока, к которому прикреплен хук. Данный хук является уведомительным, оба параметра - и *wParam*, и *lParam* - равны нулю.

WH_GETMESSAGE

- Windows вызывает этот хук перед выходом из функций **GetMessage** и **PeekMessage**. Фильтрующие функции получают указатель на структуру с сообщением, которое затем (вместе со всеми изменениями) посылается приложению, вызвавшему **GetMessage** или **PeekMessage**.
- В *lParam* находится указатель на структуру MSG:

Регистрационные хуки

- Регистрационные хуки (journal hooks) используются для записи и воспроизведения событий. Они могут устанавливаться только как системные, и, следовательно, должны использоваться как можно реже. Эти хуки воздействуют на все приложения Windows; хотя десктоп и не позволяет такого другим хукам, регистрационные хуки могут записывать и воспроизводить последовательности событий и от десктопа, и для десктопа. Другой побочный эффект регистрационных хуков в том, что все системные входные очереди проходят через один поток, который установил такой хук.
- В Win32 предусмотрена специальная последовательность действий, с помощью которой пользователь может убрать регистрационный хук (например, в случае, если он завесил систему). Windows отключит записывающий или воспроизводящий регистрационный хук, когда пользователь нажмет CTRL+ESC, ALT+ESC, или CTRL+ALT+DEL.
- Windows оповестит приложение, установившее этот хук, посылкой ему сообщения WM_CANCELJOURNAL.

WM_CANCELJOURNAL

- Это сообщение посылается с хэндлом окна, равным NULL, чтобы оно не попало в оконную процедуру. Лучший способ получить это сообщение - прикрепить к WH_GETMESSAGE фильтрующую функцию, которая бы следила за входящими сообщениями. В документация по Win32 упоминается, что приложение может получить сообщение WM_CANCELJOURNAL между вызовами функций **GetMessage** (или **PeekMessage**) и **DispatchMessage**.
- Хотя это и так, нет гарантий, что приложение будет вызывать эти функции, когда будет послано сообщение. Например, если приложение занято показом диалогового окна, главный цикл обработки сообщений не получит управление.
- Комбинации клавиш CTRL+ESC, ALT+ESC, и CTRL+ALT+DEL встроены в систему, чтобы пользователь всегда смог остановить регистрационный хук. Было бы неплохо, если каждое приложение, использующее регистрационные хуки, также предусматривало для пользователя способ остановки тотальной регистрации. Рекомендуемый способ - использовать код VK_CANCEL (CTRL+BREAK).

WH_JOURNALRECORD

- Windows вызывает этот хук при удалении события из системной очереди. Таким образом, фильтры этого хука вызываются для всех мышиных и клавиатурных событий, кроме тех, которые проигрываются регистрационным хуком на воспроизведение. Фильтрующие функции могут обработать сообщение (то есть, записать или сохранить событие в памяти, на диске, или и там, и там), но не могут изменять или отменять его.
- Фильтры этого хука могут находиться и внутри DLL, и в .EXE-файле. В Win32 для этого хука реализован только код HC_ACTION.

HC_ACTION

- Windows вызывает хук WH_JOURNALRECORD с этим кодом при удалении события из системной очереди. Этот код сигнализирует фильтрующей функции о том, что это событие является нормальным. В lParam при этом передается указатель на структуру EVENTMSG. Обычная процедура записи состоит в сохранении всех пришедших хуку структур EVENTMSG в памяти или на диске.
- Структура EVENTMSG описана в WINDOWS.H следующим образом:

```
typedef struct tagEVENTMSG {
 UINT message;
 UINT paramL;
 UINT paramH;
DWORD time;
 HWND hwnd;
 } EVENTMSG;
```

- Элемент *message* является идентификатором сообщения, одним из значений WM_*. Значения *paramL* и *paramH* зависят от источника события - мышь это или клавиатура. Если это событие мыши, в *paramL* и *paramH* передаются координаты x и y события. Если это клавиатурное событие, в *paramL* находятся два значения: скан-код клавиши в HIBYTE и виртуальный код клавиши в LOBYTE, а paramH содержит число повторений. 15-й бит числа повторений служит индикатором дополнительной клавиши. В элементе *time* хранится системное время (наступления события), которое возвращается функцией **GetTickCount**. *hwnd* - это хэндл окна, получившего событие.
- Промежуток времени между событиями определяется сравнением элементов *time* этого события с элементом *time* последующего события. Разница во времени нужна для корректного проигрывания записанных событий.

WH_JOURNALPLAYBACK

- Этот хук используется для посылки Windows клавиатурных и мышиных сообщений таким образом, как будто они проходят через системную очередь. Основное назначение этого хука - проигрывание событий, записанных с помощью хука WH_JOURNALRECORD, но его можно также с успехом использовать для посылки сообщений другим приложениям. Когда к этому хуку прикреплены фильтрующие функции, Windows вызывает первый фильтр в цепочке, чтобы получить событие. Windows игнорирует движения мыши, пока в системе установлен хук WH_JOURNALPLAYBACK. Все остальные события от клавиатуры и мыши сохраняется до тех пор, пока у хука WH_JOURNALPLAYBACK не останется функций-фильтров. Фильтры для этого хука могут располагаться как в DLL, так и в .EXE-файле. Фильтры этого хука должны знать о существовании следующих кодов:
- HC_GETNEXT
- HC_SKIP

HC_GETNEXT

- Windows вызывает WH_JOURNALPLAYBACK с этим кодом, когда получает доступ к входной очереди потока. В большинстве случаев Windows посылает этот код несколько раз для одного и того же сообщения. В *lParam* фильтру передается указатель на структуру **EVENTMSG** (см. выше). Фильтрующая функция должна занести в эту структуру код сообщения *message*, *paramL*, и *paramH*. Обычно эти значения копируются из структур, записанных ранее с помощью хука WH_JOURNALRECORD.
- Фильтрующая функция должна сообщить Windows когда нужно начинать обработку посланного сообщения. Windows необходимо для этого два значения: (1) период времени, на которое Windows должно задержать обработку сообщения; либо (2) точное время, когда это сообщение должно быть обработано. Обычно время ожидания обработки вычисляется как разница элементов *time* структуры **EVENTMSG** предыдущего сообщения и элемента *time* той же структуры текущего сообщения. Такой прием позволяет проигрывать сообщения на той же скорости, на которой они были записаны. Если сообщение необходимо проиграть немедленно, функция должна вернуть значение периода времен, равное нулю.

- Точное значение времени, в которое нужно обработать сообщение, обычно вычисляется сложением времени, которое Windows должна подождать до начала обработки сообщения и текущего системного времени, получаемого функцией **GetTickCount**. Для немедленного проигрывания сообщения используйте значение, возвращаемое функцией **GetTickCount**.
- Если система не находится в активном состоянии, Windows использует значения, переданные фильтром, для обработки события. Если система находится в активном состоянии, Windows проверяет системную очередь. Каждый раз, когда она это делает, Windows запрашивает то же самое событие с кодом HC_GETNEXT. Каждый раз, когда функция-фильтр получает код HC_GETNEXT, она должна вернуть новое значение времени ожидания, принимая во внимание время, прошедшее между вызовами функций. Элементы *message*, *paramH* и *paramL*, скорее всего, не потребуют изменений между вызовами.

HC_SKIP

- Windows вызывает хук WH_JOURNALPLAYBACK после окончания обработки сообщения, полученного от WH_JOURNALPLAYBACK. Это происходит в момент мнимого удаления события из системной очереди (мнимой, так как событие не находилось в системной очереди, а было сгенерировано хуком WH_JOURNALPLAYBACK).
- Этот код хука сигнализирует фильтрующей функции о том, что событие, возвращенное фильтром во время вызова предыдущего HC_GETNEXT, попало в приложение. Фильтрующая функция должна приготовиться вернуть следующее событие по приходу кода HC_GETEVENT. Когда фильтрующая функция определяет, что больше нечего проигрывать, она должна удалиться из цепочки фильтров хука во время обработки кода HC_SKIP.

WH_KEYBOARD

- Windows вызывает этот хук когда функции **GetMessage** или **PeekMessage** собираются вернуть сообщения WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN, или WM_CHAR. Когда хук установлен как локальный, эти сообщения должны поступать из входной очереди потока, к которому прикреплен хук. Фильтрующая функция получает виртуальный код клавиши и состояние клавиатуры на момент вызова клавиатурного хука. Фильтры имеют возможность отменить сообщение. Фильтрующая функция, прикрепленная к этому хуку, должна знать о существовании следующих кодов:
- HC_ACTION
- HC_NOREMOVE

- **HC_ACTION**
- Windows вызывает хук WH_KEYBOARD с этим кодом при удалении события из системной очереди.
- **HC_NOREMOVE**
- Windows вызывает хук WH_KEYBOARD с этим кодом, когда клавиатурное сообщение не удаляется из очереди, потому что приложение вызвало функцию **PeekMessage** с параметром PM_NOREMOVE. При вызове хука с этим кодом не гарантируется передача действительного состояние клавиатуры. Приложение должно знать о возможности возникновения подобной ситуации.

WH_MOUSE

- Windows вызывает этот хук после вызова функций **GetMessage** или **PeekMessage** при условии наличия сообщения от мыши. Подобно хуку WH_KEYBOARD фильтрующие функции получают код - индикатор удаления сообщения из очереди (HC_NOREMOVE), идентификатор сообщения мыши и координаты x и y курсора мыши.
- Фильтры имеют возможность отменить сообщение. Фильтры для этого хука должны находиться в DLL.

WH_MSGFILTER

- Windows вызывает этот хук, когда диалоговое окно, информационное окно, полоса прокрутки или меню получают сообщение, либо когда пользователь нажимает комбинацию клавиш ALT+TAB (или ALT+ESC) при активном приложении, установившем этот хук. Данный хук устанавливается для конкретного потока, поэтому его безопасно размещать как в самом приложении, так и в DLL. Фильтрующая функция этого хука получает следующие коды:
- MSGF_DIALOGBOX: Сообщение предназначено либо диалоговому, либо информационному окну.
- MSGF_MENU: Сообщение предназначено меню.
- MSGF_SCROLLBAR: Сообщение предназначено полосе прокрутки.
- MSGF_NEXTWINDOW: Происходит переключение фокуса на следующее окно.
- В WINUSER.H определено больше MSGF_-кодов, но в настоящее время они не используются хуком WH_MSGFILTER.
- В *lParam* передается указатель на структуру, содержащую информацию о сообщении. Хуки WH_SYSMSGFILTER вызываются перед хуками WH_MSGFILTER. Если какая-нибудь из фильтрующих функций хука WH_SYSMSGFILTER возвратит TRUE, хуки WH_MSGFILTER не будут вызваны.

WH_SHELL

- Windows вызывает этот хук при определенных действиях с окнами верхнего уровня - top-level windows (то есть, с окнами, не имеющими владельца). Когда хук установлен как локальный, он вызывается только для окон, принадлежащих потоку, установившему хук. Этот хук является информирующим, поэтому фильтры не могут изменять или отменять событие. В *wParam* передается хэндл окна; параметр *lParam* не используется. Для данного хука в WINUSER.H определены три кода:
- HSHELL_WINDOWCREATED: Windows вызывает хук WH_SHELL с этим кодом при создании окна верхнего уровня. Когда фильтр получает управление, это окно уже создано.
- HSHELL_WINDOWDESTROYED: Windows вызывает хук WH_SHELL с этим кодом перед удалением окна верхнего уровня.
- HSHELL_ACTIVATESHELLWINDOW: На данный момент этот код не используется.

WH_SYSMSGFILTER

- Этот хук идентичен хуку WH_MSGFILTER за тем исключением, что он имеет системную область видимости. Windows вызывает этот хук, когда диалоговое окно, информационное окно, полоса прокрутки или меню получает сообщение, либо когда пользователь нажимает комбинации клавиш ALT+TAB или ALT+ESC. Фильтр получает те же коды, что и фильтры хука WH_MSGFILTER.
- В *lParam* передается указатель на структуру, содержащую информацию о сообщении. Хуки WH_SYSMSGFILTER вызываются до хуков WH_MSGFILTER. Если любая из фильтрующих функций хука WH_SYSMSGFILTER вернет TRUE, фильтры хука WH_MSGFILTER не будут вызваны.

Внедрение DLL и перехват API-вызовов

- О среде Windows каждый процесс получает свое адресное пространство.
- Указатели, используемые Вами для ссылки на определенные участки памяти, — это адреса в адресном пространстве Вашего процесса, и в нем нельзя создать указатель, ссылающийся на память, принадлежащую другому процессу.
- Так, если в Вашей программе есть «жучок», из-за которого происходит запись по случайному адресу, он не разрушит содержимое памяти, отведенной другим процессам.

- Раздельные адресные пространства очень выгодны и разработчикам, и пользователям. Первым важно, что Windows перехватывает обращения к памяти по случайным адресам, вторым — что операционная система более устойчива и сбой одного приложения не приведет к краху другого или самой системы.
- Но, конечно, за надежность приходится платить: написать программу, способную взаимодействовать с другими программами или манипулировать другими процессами, теперь гораздо сложнее.
- Вот ситуации, в которых требуется прорыв за границы процессов и доступ к адресному пространству другого процесса:

1. создание подкласса окна, порожденного другим процессом;
2. получение информации для отладки (например, чтобы определить, какие DLL используются другим процессом);
3. установка ловушек (hooks) в других процессах.

- Порождение подкласса окна, созданного чужим процессом, возможно.
- Проблема не столько в создании подкласса, сколько в закрытости адресного пространства процесса.
- Если бы можно было как-то поместить код своей оконной процедуры в адресное пространство процесса A, это позволило бы вызвать *SetWindowLongPtr* и передать ей адрес *MySubclassProc,* в процессе A.
- Называется такой прием внедрением (injecting) DLL в адресное пространство процесса. Существует несколько способов подобного внедрения

Внедрение DLL с использованием реестра

- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows_NT\CurrentVersion\Windows\AppInit_DLLs

- Значением параметра AppInit_DLLs может быть как имя одной DLL (с указанием пути доступа), так и имена нескольких DLL, разделенных пробелами или запятыми.
- Поскольку пробел используется здесь в качестве разделителя, в именах файлов не должно быть пробелов. Система считывает путь только первой DLL в списке — пути остальных DLL игнорируются, поэтому лучше размещать свои DLL в системном каталоге Windows, чтобы не указывать пути.

- При следующей перезагрузке компьютера Windows сохранит значение этого параметра. Далее, когда User32.dll будет спроецирован на адресное пространство процесса, этот модуль получит уведомление DLL_PROCESS_ ATTACH и после его обработки вызовет *LoadLibrary* для всех DLL, указанных в параметре AppInit_ DLLs.
- В момент загрузки каждая DLL инициализируется вызовом ее функции DllMain с параметром fwdReason, равным DLL_PROCESS_ATTACH. Поскольку внедряемая DLL загружается на такой ранней стадии создания процесса, будьте особенно осторожны при вызове функций. Проблем с вызовом функций Kernel32. dll не должно быть, но в случае других DLL они вполне вероятны — User32. dll не проверяет, успешно ли загружены и инициализированы эти DLL.
- Это простейший способ внедрения DLL. Все, что от вас требуется, — добавить значение в уже существующий параметр реестра.

Недостатки способа

1. Ваша DLL проецируется на адресные пространства только тех процессов, на которые спроецирован и модуль User32.dll. Его используют все GUI-приложения, но большинство программ консольного типа — нет. Поэтому такой метод не годится для внедрения DLL, например, в компилятор или компоновщик.
2. Ваша DLL проецируется на адресные пространства всех GUI-процессов. Но Вам-то почти наверняка надо внедрить DLL только в один или несколько определенных процессов. Чем больше процессов попадет "под тень" такой DLL, тем выше вероятность аварийной ситуации. Ваш код выполняется потоками этих процессов, и, если он зациклится или некорректно обратится к памяти, Вы повлияете на поведение и устойчивость соответствующих процессов.

Поэтому лучше внедрять свою DLL в как можно меньшее число процессов.

Недостатки способа

3. Ваша DLL проецируется на адресное пространство каждого GUI-процесса в течение всей его жизни.

- Тут есть некоторое сходство с предыдущей проблемой. Желательно не только внедрять DLL в минимальное число процессов, но и проецировать ее на эти процессы как можно меньшее время.
- Так что лучшее решение — внедрять DLL только на то время, в течение которого она действительно нужна конкретной программе.

Внедрение DLL с помощью ловушек

- Внедрение DLL в адресное пространство процесса возможно и с применением ловушек. Чтобы они работали так же, как и в 16-разрядной Windows, Microsoft пришлось создать механизм, позволяющий внедрять DLL в адресное пространство другого процесса

Рассмотрим его на примере

- Процесс А (вроде утилиты Spy++) устанавливает ловушку WH_GETMESSAGE и наблюдает за сообщениями, которые обрабатываются окнами в системе. Ловушка устанавливается вызовом *SetWindowsHookEx*

- HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hInstDll, 0);

Как все это действует:

- 1. Поток процесса B собирается направить сообщение какому-либо окну.
- 2. Система проверяет, не установлена ли для данного потока ловушка WH_GETMESSAGE.
- 3. Затем выясняет, спроецирована ли DLL, содержащая функцию *GetMsgProc,* на адресное пространство процесса B.
- 4. Если указанная DLL еще не спроецирована, система отображает ее на адресное пространство процесса B и увеличивает счетчик блокировок (lock count) проекции DLL в процессе B на 1.

- 5. Система проверяет, не совпадают ли значения *hinstDll* этой DLL, относящиеся к процессам A и B Если *hinstDll* в обоих процессах одинаковы, то и адрес *GetMsgProc* в этих процессах тоже одинаков Тогда система может просто вызвать *GetMsgProc* в адресном пространстве процесса A. Если же *hinstDll* различны, система определяет адрес функции *GetMsgProc* в адресном пространстве процесса B по формуле:

GetMsgProc B = histDll B + (GetMsgProc A - hinstDll A)

Вычитая *hinstDll* из *GetMsgProcA,* Вы получаете смещение (в

байтах) адреса функции *GetMsgProc.*

Добавляя это смещение к *hinstDll B,* Вы получаете адрес *GetMsgProc,* соответствующий проекции DLL в адресном пространстве процесса B

- 6. Счетчик блокировок проекции DLL в процессе B увеличивается на 1.
- 7. Вызывается *GetMsgProc* в адресном пространстве процесса B.
- 8. После возврата из *GetMsgProc* счетчик блокировок проекции DLL в адресном пространстве процесса B уменьшается на 1.

- Когда система внедряет или проецирует DLL, содержащую функцию фильтра ловушки, проецируется вся DLL, а не только эта функция. А значит, потокам, выполняемым в контексте процесса B, теперь доступны все функции такой DLL.
- Итак, чтобы создать подкласс окна, сформированного потоком другого процесса, можно сначала установить ловушку WH_GETMESSAGE для этого потока, а затем — когда будет вызвана функция *GetMsgProc* - обратиться к *SetWtndowLongPtr* и создать подкласс Разумеется, процедура подкласса должна быть в той же DLL, что и *GetMsgProc.*
- В отличие от внедрения DLL с помощью реестра этот способ позволяет в любой момент отключить DLL от адресного пространства процесса, для чего достаточно вызвать:

- BOOL UnhookWindowsHookEx(HHOOK hHook);

- Когда поток обращается к этой функции, система просматривает внутренний список процессов, в которые ей пришлось внедрить данную DLL, и уменьшает счетчик ее блокировок на 1 Как только этот счетчик обнуляется, DLL автоматически выгружается. Вспомните: система увеличивает его непосредственно перед вызовом *GetMsgProc* (см. выше п. 6). Это позволяет избежать нарушения доступа к памяти Если бы счетчик не увеличивался, то другой поток мог бы вызвать *UnhookWindowsHookEx* в тот момент, когда поток процесса В пытается выполнить код *GetMsgProc,*
- Все это означает, что нельзя создать подкласс окна и тут же убрать ловушку — она должна действовать в течение всей жизни подкласса.

Внедрение DLL с помощью удаленных потоков

- Третий способ внедрения DLL — самый гибкий. В нем используются многие особенности Windows: процессы, потоки, синхронизация потоков, управление виртуальной памятью, поддержка DLL и Unicode.
- Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом
- Изначально многие из них были рассчитаны на применение в отладчиках и других инструментальных средствах. Но ничто не мешает использовать их и в обычном приложении.

- Внедрение DLL этим способом предполагает вызов функции *LoadLibrary* потоком целевого процесса для загрузки нужной DLL.
- Так как управление потоками чужого процесса сильно затруднено, Вы должны создать в нем свой поток.
- Windows-функция *CreateRemoteThread* делает эту задачу несложной:

```
HANDLE CreateRemoteThread( HANDLE hProcess,
 PSECURITY_ATTRIBUTES psa,
 DWORD dwStackSize,
 PTHREAD_START_ROUTINE pfnStartAddr,
 PVOID pvParam,
 DWORD fdwCreate,
PDWORD pdwThreadId);
```

- Имеет дополнительный параметр *hProcess,* идентифицирующий процесс, которому будет принадлежать новый поток. Параметр *pfnStartAddr* определяет адрес функции потока.
- Этот адрес, разумеется, относится к удаленному процессу — функция потока не может находиться в адресном пространстве Вашего процесса.

- Теперь создан поток в другом процессе. Но как заставить этот поток загрузить нашу DLL?
- Нужно, чтобы он вызвал функцию *LoadLibrary:*

HINSTANCE LoadLibrary(PCTSTR pszlibFile);

- Заглянув в заголовочный файл WinBase.h, Вы увидите, что для *LoadLibrary* там есть такие строки:

- HINSTANCE WINAPI LoadLibraryA(LPCSTR pszLibFileName);
- HINSTANCE WINAPI LoadLibraryW(LPCWSTR pszLibFileName);

- В действительности существует две функции *LoadLibrary LoadLibraryA* и *LoadLibraryW.*
- Они различаются только типом передаваемого параметра. Если имя фаЙла библиотеки хранится как ANSI-строка, вызывайте *LoadLibraryA;* если же имя файла представлено Unicode-строкой — *LoadLibraryW.*
- Самой функции *LoadLibrary* нет.
- В большинстве программ макрос *LoadLibrary* раскрывается в *LoadLibraryA.*

- По сути, требуется выполнить примерно такую строку кода
- HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, LoadlibraryA, "C:\\MyLib.dll", 0, NULL);

- Или, если Вы предпочитаете Unicode
- HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, LoadLibraryW, L"C :\\MyLib.dll" , 0, NULL);

- Новый поток в удаленном процессе немедленно вызывает *LoadLibraryA* (или *LoadLibraryW),* передавая ей адрес полного имени DLL.

Так что *CreateRemoteThread* надо вызвать так:

```
// получаем истинный адрес LoadLibraryA в Kernel32 dll
   PTHREAD_START_ROUTIHE pfnThreadRtn =
   (PTHREAD_START_ROUTINE)
GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
   "LoadLibraryA");
HANDLE hThread = CreateRemoteThread(hProcessRemote,
   NULL, 0, pfnThreadRtn, "C.\\MyLib.dll", 0, NULL);
```

Или, если Вы предпочитаете Unicode:

```
// получаем истинный адрес LoadLibraryA в Kernel32.dll
   PTHREAD_START_ROUTINE pfnThreadRtn =
   (PTHREAD_START ROUTINE)
GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
   "LoadLibraryW");
HANDLE hThread = CreateRemoteThread(hProcessRemote,
   NULL, 0, pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

- Проблема связана со строкой, в которой содержится полное имя файла DLL. Строка «C:\\MyLib.dll» находится в адресном пространстве вызывающего процесса.
- Ее адрес передается только что созданному потоку, который в свою очередь передает его в *LoadLibraryA.*
- Но, когда *LoadLibraryA* будет проводить разыменование (dereferencing) этого адреса, она не найдет по нему строку с полным именем файла DLL и скорее всего вызовет нарушение доступа в потоке удаленного процесса;
- пользователь увидит сообщение о необрабатываемом исключении, и удаленный процесс будет закрыт.
- Все верно: Вы благополучно угробили чужой процесс, сохранив свой в целости и сохранности.

- Эта проблема решается размещением строки с полным именем файла DLL в адресном пространстве удаленного процесса.
- Впоследствии, вызывая *CreateRemoteThread,* мы передадим ее адрес (в удаленном процессе).
- На этот случай в Windows предусмотрена функция *VirtualAllocEx,* которая позволяет процессу выделять память в чужом адресном пространстве:

```
PVOID VirtualAllocEx( HANDLE hProcess, PVOIO
    pvAddress, SIZE_T dwSize, DWORD flAllocationType,
    DWORD flProtect);
```

- А освободить эту память можно с помощью функции *VirtualFreeEx.*

```
BOOL VirtualFreeEx( HANDLE hProcess, PVOID pvAddress,
    SIZE_T dwSize, DWORD dwFreeType);
```

- Выделив память, мы должны каким-то образом скопировать строку из локального адресного пространства в удаленное.
- Для этого в Windows есть две функции

```
BOOL ReadProcessMemory( HANDLE hProcess, PVOID
    pvAddressRemote, PVOID pvBufferLocal, DWORD dwSize, PDWORD
    pdwNumBytesRead);

BOOL WriteProcessMemory( HANDLE hProcess, PVOID
    pvAddressRemote, PVOTD pvBufferLocal, DWOHD dwSize,
    PDWORD pdwNumBytesWritten);
```

- Параметр *hProcess* идентифицирует удаленный процесс, *pvAddressRemote* и *pvBufferLocal* определяют адреса в адресных пространствах удаленного и локального процесса, а *dwSize* — число передаваемых байтов.
- По адресу, на который указывает параметр *pdwNumBytesRead* или *pdwNumBytesWritten,* возвращается число фактически считанных или записанных байтов.

Итак:

1. Выделите блок памяти в адресном пространстве удаленного процесса через *VirtualAllocEx.*
2. Вызвав *WriteProcessMemory,* скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п. 1
3. Используя *GetProcAddress,* получите истинный адрес функции *LoadLibraryA* или *LoadLibraryW* внутри Kernel32.dll.
4. Вызвав *CreateRemoteThread,* создайте поток в удаленном процессе, который вызовет соответствующую функцию *LoadLibrary,* передав ей адрес блока памяти, выделенного в п. 1.

- Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве.
- Для очистки после завершения удаленного потока потребуется несколько дополнительных операций.

1. Вызовом *VirtualFreeEx* освободите блок памяти, выделенный в п. 1.
2. С помощью *GetProcAddress* определите истинный адрес функции *FreeLibrary* внутри Kernel32.dll.
3. Используя *CreateRemoteThtead,* создайте в удаленном процессе поток, который вызовет *FreeLibrary c* передачей HINSTANCE внедренной DLL.

Перехват API-вызовов подменой кода

1. Найдите адрес функции, вызов которой вы хотите перехватывать (например, *ExitProcess в Kernel32.dll).*
2. Сохраните несколько первых байтов этой функции в другом участке памяти. На их место вставьте машинную команду JUMP для перехода по адресу подставной функции. Естественно, сигнатура вашей функции должна быть такой же, как и исходной, т. е. все параметры, возвращаемое значение и правила вызова должны совпадать. Теперь, когда поток вызовет перехватываемую функцию, команда JUMP перенаправит его к вашей функции. На этом этапе вы можете выполнить любой нужный код.
3. Снимите ловушку, восстановив ранее сохраненные (в п. 2) байты.

Если теперь вызвать перехватываемую функцию (таковой больше не являющуюся), она будет работать так, как работала до установки ловушки.

Основы Native API

Native приложения

- Программы, предназначенные для выполнения на операционных системах Windows, способные запускаться на раннем этапе загрузки Windows, до окна входа в систему и даже до запуска каких-либо подсистем Windows.
- Экран при загрузке Windows, в котором, например, происходит проверка диска и есть тот самый режим.
- Native приложения используют только Native API, они могут использовать только функции, экспортируемые из библиотеки ntdll.dll. Для них недоступны функции WinAPI.

Native приложения

- Native приложения запускаются на экране, который возникает до появления окна входа в систему.
- Примером native приложения является приложение chkdsk, которое запускается перед входом в Windows, если предварительно была запущена проверка системного раздела на ошибки и отложена до перезагрузки.
- Приложение работает, выводя сообщения экран, а затем происходит обычный запуск Windows.

- Преимущества использования этого режима: большая часть компонентов Windows ещё не запущена, отсутствуют многие ограничения. Этот режим, например, используется в приложениях, которые хотят что-то сделать с системным разделом Windows, но не могут, пока запущена операционная система: дефрагментаторы, конверторы файловой системы, и тому подобные утилиты.

Что нужно знать:

- Native приложения компилируются с помощью WDK - *Windows Driver Kit* (также известный, как DDK). Есть возможность делать их и в какой-то другой среде разработки, но в WDK проще всего.
- Native приложения используют *Native API*. Оно частично документировано в MSDN для использования при написании драйверов. Но документированы не все функции. Информацию по остальным нужно брать из неофициальных источников. Например, на сайте http://undocumented.ntinternals.net/
- Функции в *ntdll.dll* имеют префиксы **Zw** и **Nt**, а также некоторые другие. Видно, что у Zw и Nt функции дублируются названия. На самом деле это одни и те же функции. Если искать в сети пример использования какой-либо функции, стоит поискать сначала с одним префиксом, потом с другим, иначе можно что-то упустить.

Что нужно знать:

- Функции в *ntdll.dll* имеют префиксы **Zw** и **Nt**, а также некоторые другие. Видно, что у Zw и Nt функции дублируются названия. На самом деле это одни и те же функции.
- Если искать в сети пример использования какой-либо функции, стоит поискать сначала с одним префиксом, потом с другим, иначе можно что-то упустить.
- Для программирования нужны прототипы функций Native API, но в заголовочных файлах WDK присутствуют не все определения.
- Нужно использовать альтернативные заголовочные файлы, содержащие в том числе и определения недокументированных функций и типов данных.

Что нужно знать:

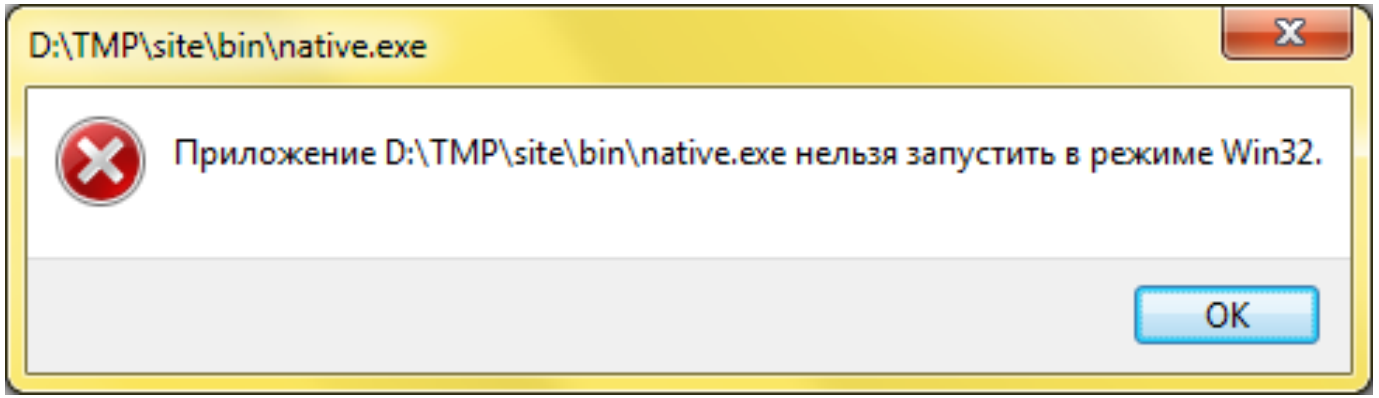
- Программировать на чистом Native API неудобно. Не обойтись без библиотеки, в которой уже реализованы некоторые рутинные действия. Существует библиотека с открытым кодом - ZenWINX, можно пользоваться ей.
- Чтобы native приложение запустилось при запуске Windows, надо положить его в каталог system32, а в ключ реестра HKLM\System\CurrentControlSet\Control\Session Manager\BootExecute прописать его имя файла, и аргументы, если они есть.
- Ключ имеет тип MULTI_SZ, может содержать несколько строк.

Что нужно знать:

- Программа, прописанная в этом ключе, имеет свойство запускаться даже в безопасном режиме Windows (safe mode), так что нужно быть осторожным. Ошибка в программе - и система не запустится.
- Но можно внутри приложения отслеживать факт запуска в safe mode и обрабатывать этот режим отдельно, например сделать завершение программы, если она обнаружила себя запущенной в safe mode.
- Кроме того, несмотря на то, что программа запускается и может выполнять какие-то действия, в этом режиме не работает вывод на консоль. Невозможно взаимодействие с пользователем. Это следует учитывать.

Точка входа

- У native приложений точка входа не main и не wmain, а **NtProcessStartup**. В PE-заголовке EXE-файла есть специальное поле, означающее подсистему, в которой выполняется приложение.
- У native приложений в это поле установлено специальное значение, означающее, что EXE не требует подсистемы. У обычных приложений ставится значение, соответствующее подсистемам "Windows GUI" или "Windows console".
- Native приложения не запускаются в обычном режиме работы Windows.

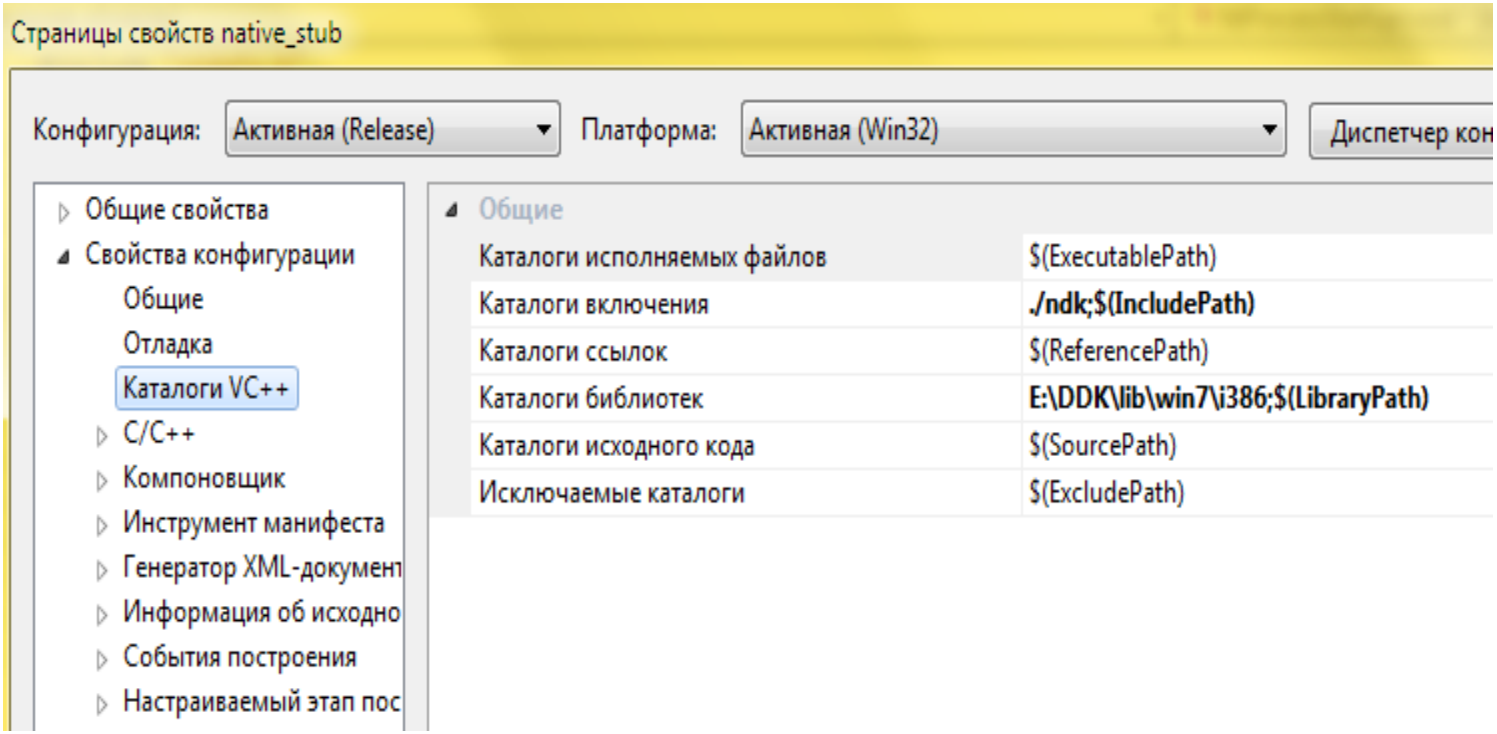


- Разработка **Native API приложений** обычно происходит с помощью компилятора, входящего в состав Windows Driver Kit (WDK). Однако, можно обойтись и без WDK. Visual Studio тоже способна создавать приложения, не требующие функционирования подсистемы Win32 для запуска.
- Для этого понадобится сама Visual Studio, заголовочные файлы и файл **ntdll.lib**, который придётся достать (из WDK или из сети ☺).

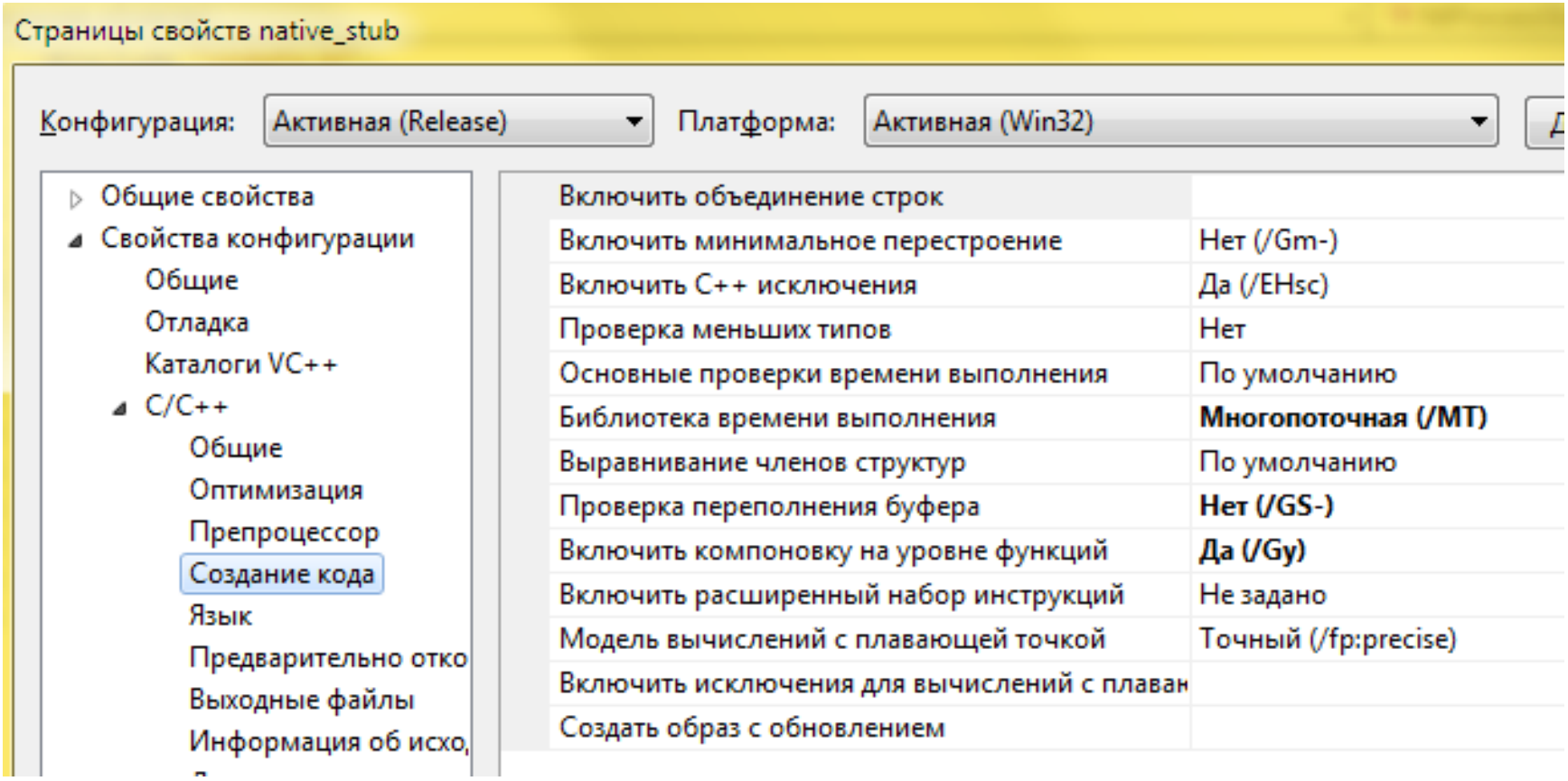
Настройка проекта Visual Studio

- В Visual Studio нужно создать новый проект — консольное приложение.
- Это оптимальный тип проекта для переделки в Native-приложение.
- В каталог с проектом или в иное место нужно положить заголовочные файлы .

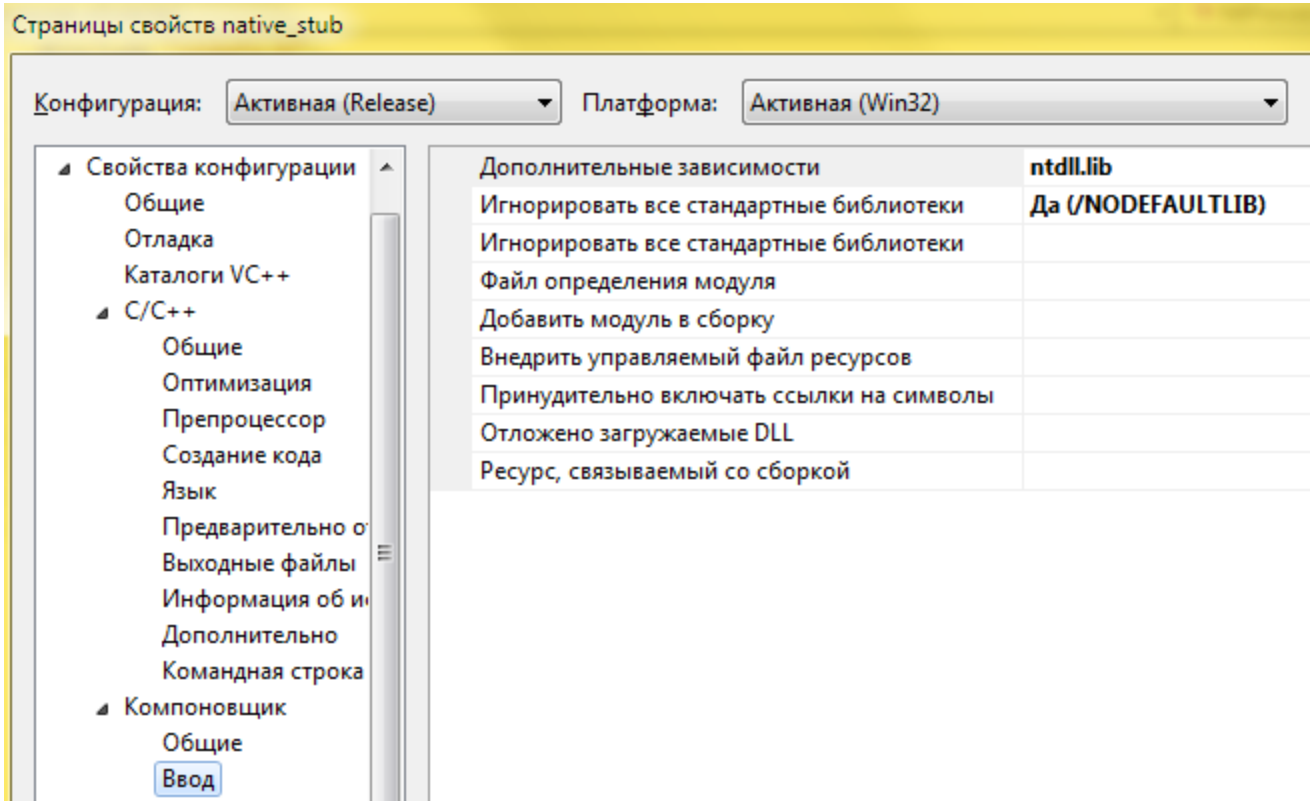
В настройках проекта во вкладке «Свойства конфигурации › Каталоги VC++» нужно добавить путь к NDK в пункт «Каталоги включения», и путь к ntdll.lib в пункте «Каталоги библиотек».



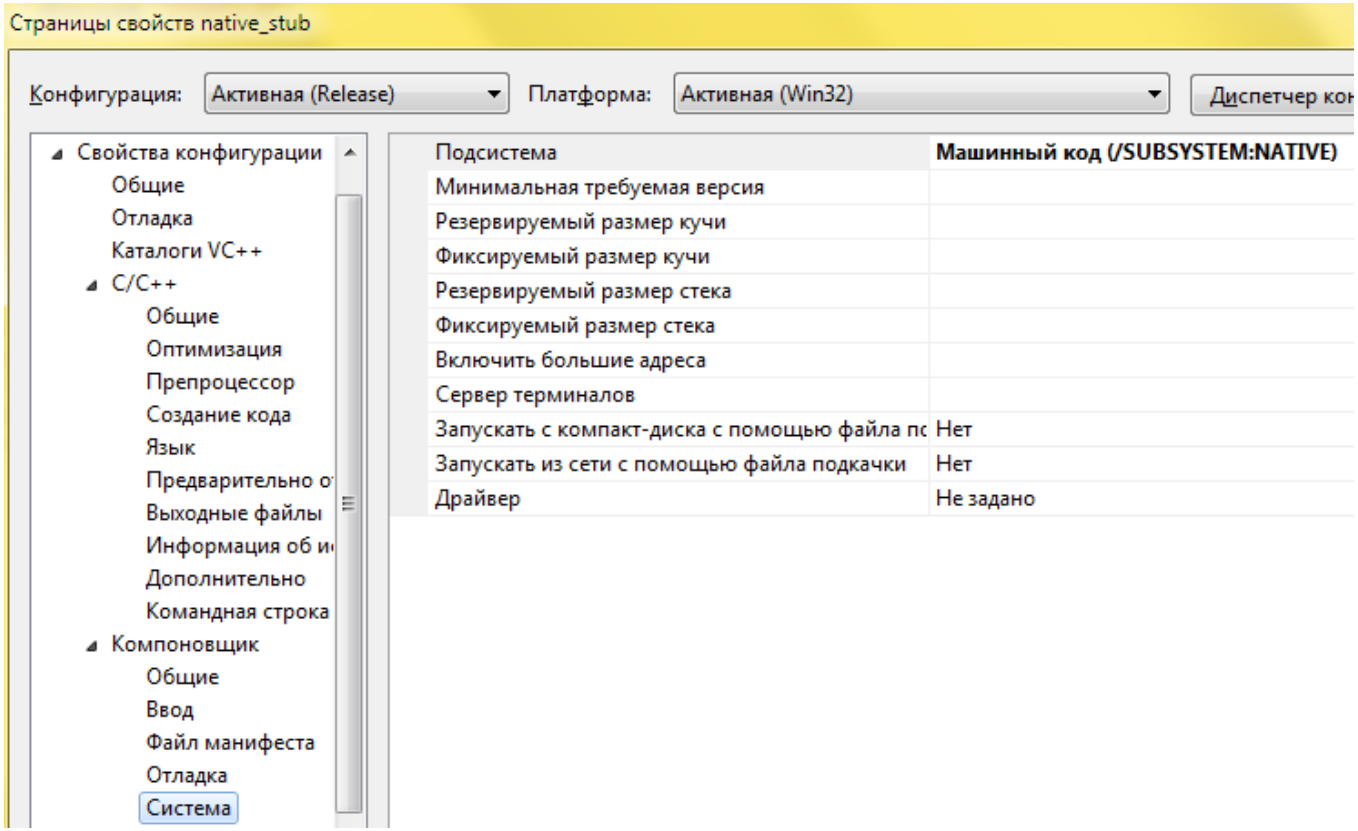
- Запуск Native-приложения в отладчике Visual Studio, да и вообще, при работающей системе невозможен. Так что сразу можно выбирать тип сборки проекта «Release».
- На вкладке «C/C++ › Создание кода» нужно отключить проверку переполнения буфера (опция должна быть выставлена в /GS-). Если этого не сделать, то при сборке Native-приложения возникнет ошибка:



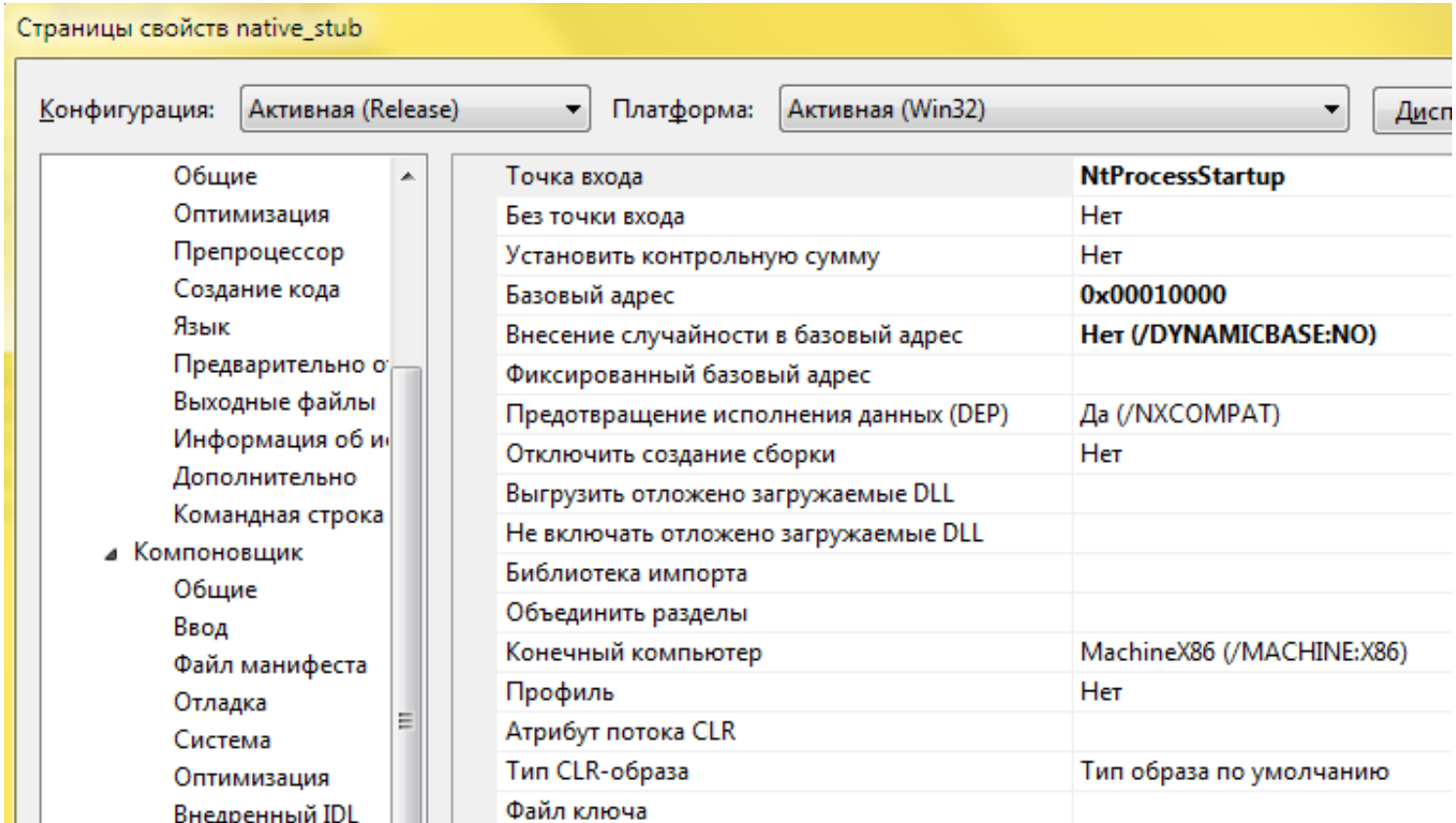
- На вкладке «Компоновщик › Ввод» в пункте «Дополнительные зависимости» нужно убрать всё, что содержится в нём и добавить только одну зависимость: ntdll.lib.
- В пункте «Игнорировать все стандартные библиотеки» нужно выбрать «Да». Native-приложение не может позволить себе иметь зависимость от других библиотек, кроме ntdll, по причине того, что все остальные библиотеки рассчитаны на работу в рамках подсистемы Win32.
- Native-приложение работает вне этой системы.



- На вкладке «Компоновщик › Система» в пункте «Подсистема» нужно выбрать «Машинный код (**/SUBSYSTEM:NATIVE**)».



- На вкладке «Компоновщик › Дополнительно» в пункте «Точка входа» нужно написать **NtProcessStartup**. Именно так обычно называется точка входа в Native-приложение, а не main и не WinMain.
- В исходном тексте, который создала Visual Studio для нашего проекта нужно убрать сгенерированную ей точку входа и написать свою.
- В пункте «Базовый адрес» нужно написать «0x00010000».



Точка входа Native-приложения

- В файле native_stub.cpp, являющимся главным файлом проекта, должна содержаться точка входа, прописанная ранее в настройках проекта. Нужно удалить сгенерированную точку входа main и написать свою, под именем NtProcessStartup:

```
void NtProcessStartup(void* StartupArgument)
{
  UNICODE_STRING str;
  PPEB pPeb = (PPEB)StartupArgument;
  RtlNormalizeProcessParams(pPeb->ProcessParameters);
  RtlInitUnicodeString(&str, L"Hello, world!\nCommand line is: ");
  NtDisplayString(&str);
  RtlInitUnicodeString(&str, pPeb->ProcessParameters-
    >CommandLine.Buffer);
  NtDisplayString(&str);
  NtTerminateProcess(NtCurrentProcess(), 0);
}
```

- Программа выводит «Hello, world!», показывает содержимое своей командной строки и завершает своё выполнение вызовом **NtTerminateProcess**.

Заголовочный файл проекта

- В stdafx.h нужно написать следующее:

```
#pragma once
#define WIN32_NO_STATUS
#include <windows.h>
#include <ntndk.h>
```

Запуск

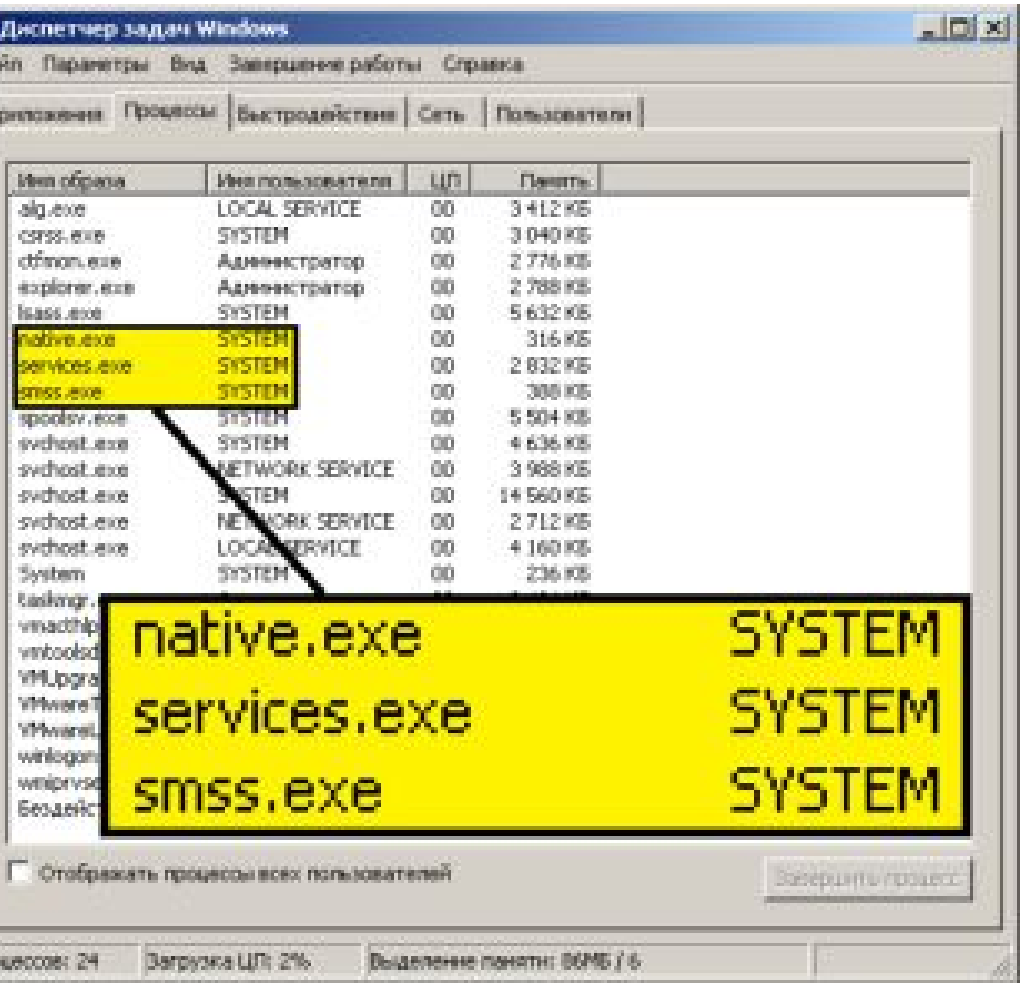
- В результате сборки проекта получается файл *.exe размером 2 Кб.
- Антивирус может ошибочно считать его руткитом (TR/Rootkit.Gen), хотя файл не содержит ничего, кроме вывода своей командной строки на экран.
- Запуск файла возможен стандартным для Native-приложений способом: через ключ реестра BootExecute.

Запуск приложений через ключ реестра BootExecute

- **Автозапуск приложений** режима native (задаётся в ветке реестра **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager**.
- Там есть ключ, позволяющий запустить приложение на этапе загрузки системы - **BootExecute**.
- Это мультистроковый параметр, содержащий строку «autocheck autochk *». После неё можно добавить свою команду запуска. Например, можно поместить native.exe в папку %systemroot%\system32, а в BootExecute прописать строку «native». В результате native.exe запустится сразу после autochk.exe при запуске системы. Здесь же можно указать командную строку процесса, например «native -some -command». Чтобы запустить программу из любого каталога системы, нужно указать полный путь к исполняемому файлу, в обычном формате (например, C:\tmp\native.exe).

Модификаторы режима запуска

- При указании имени native-приложения, кроме идентификатора autocheck (используется при указании autochk в этом списке) возможны идентификаторы async и debug.
- Идентификатор debug приводит к установке ProcessParameters->DebugFlags = TRUE. Идентификатор async приводит к тому, что система не ожидает завершения запускаемого процесса, и оно продолжает работать, а система в это время продолжает загрузку.
- В результате получается приложение, работающее в живой системе, отображающееся в диспетчере задач как запущенное от имени пользователя SYSTEM.



- Второй ключ реестра, через который возможен запуск, носит название **SetupExecute** и полностью аналогичен BootExecute.
- Разница между ними в том, что запуск из этих ключей происходит на разных этапах инициализации системы. На этапе запуска из SetupExecute в системе уже создан файл подкачки и инициализированы переменные среды, а на этапе BootExecute еще нет.

Запуск процесса из приложения, использующего Native API

- При разработке приложений Native API может возникнуть необходимость запустить процесс. В native режиме невозможен запуск Win32-приложений, так как процессы подсистемы Win32 при создании требуют уведомления CSRSS о новом процессе (а он ещё неактивен).
- В native режиме возможен запуск других native приложений с помощью функции RtlCreateUserProcess.
- В параметрах функции нужно в соответствующих структурах передать полный путь к исполняемому файлу, причём в NT-формате (то есть с префиксом \??\), имя файла процесса для отображения в списке процессов и командную строку с параметрами (это строка, которой был запущен процесс, содержащая его ключи запуска).

Запуск процесса

- Например, запускаем процесс **autochk.exe** с параметрами, находясь в каталоге C:\windows\system32.
- Тогда в RtlCreateUserProcess нужно будет передать следующие строки:
- Имя файла для отображения в списке процессов: **autochk.exe**
- Командная строка: **autochk.exe /p \??\C:**
- Полный путь: \??\C:\windows\system32\autochk.exe

- При попытке запустить не Native, а Win32 приложение произойдёт ошибка и вы увидите синий экран.
- Это происходит потому, что функция CreateNativeProcess в Native shell не проверяет подсистему запускаемого исполняемого файла.
- Запускать следует либо собственные native приложения, либо native приложения, идущие в комплекте Windows, такие как autocheck.exe, autoconv.exe, autofmt.exe, autolfn.exe.

Информационные ссылки

1. Начальные сведения о Native API
2. Недокументированные функции NTDLL
3. WDK https://developer.microsoft.com/ru-ru/windows/hardware/windows-driver-kit
4. NTAPI Undocumented Functions
5. ZenWINX Library http://ultradefrag.sourceforge.net/doc/lib/zenwinx/